POLITECHNIKA WARSZAWSKA

DYSCYPLINA NAUKOWA INFORMATYKA
TECHNICZNA I TELEKOMUNIKACJA /
DZIEDZINA NAUK INŻYNIERYJNO-TECHNICZNYCH

Rozprawa doktorska

mgr inż. Piotr Białczak

Wykorzystanie protokołu HTTP do identyfikacji i klasyfikacji złośliwego oprogramowania

Promotor
dr hab. inż. Wojciech Mazurczyk, profesor uczelni

WARSZAWA 2022

Streszczenie

Złośliwe oprogramowanie jest poważnym zagrożeniem współczesnego Internetu. Przestępcy używają go do wysyłania niechcianych wiadomości, wymuszania okupu przez zaszyfrowanie plików czy wykradania danych logowania do banku. Do komunikacji wykorzystywane są w nim popularne protokoły sieciowe, w tym często protokół HyperText Transfer Protocol (HTTP). Celem niniejszej rozprawy doktorskiej jest wykazanie, że żądania tego protokołu wygenerowane przez różne rodziny złośliwego oprogramowania mogą być użyte do ich identyfikacji i klasyfikacji. Do przeprowadzenia oceny eksperymentalnej stworzono zbiory danych ruchu sieciowego obejmujące 121 rodzin złośliwego oprogramowania oraz zestaw popularnych aplikacji niezłośliwych. Przeprowadzone badania podzielono na trzy części. W części pierwszej dokonano identyfikacji charakterystycznych cech żądań HTTP umożliwiających odróżnienie złośliwego oprogramowania od aplikacji niezłośliwych. Cechy te stały się bazą do drugiej części analizy, której efektem było stworzenie narzędzia *Hfinger* umożliwiającego tworzenie unikalnych reprezentacji żądań HTTP. Reprezentacje te można wykorzystać do identyfikacji złośliwego oprogramowania przez rozróżnienie jego rodzin, a także jego konkretnych działań, np. ataków lub pobierania rozkazów. W części trzeciej skupiono się natomiast na problemie klasyfikacji złośliwego oprogramowania przy użyciu algorytmów uczenia maszynowego, tzn. przypisania nazw konkretnych rodzin do analizowanego ruchu sieciowego. Problem ten został rozszerzony o rozpoznanie obecności klas, które były nieznane w trakcie treningu klasyfikatora, czyli tzw. rozpoznawanie otwartozbiorowe (Open Set Recognition). Wykorzystano przy tym dwa sposoby reprezentacji żądań HTTP: bazujący na narzędziu *Hfinger* oraz na analizie n-gramowej. Według wiedzy autora niniejszej rozprawy jest to pierwsza praca wykorzystująca rozpoznawanie otwartozbiorowe do klasyfikacji ruchu protokołu HTTP złośliwego oprogramowania.

Słowa kluczowe: złośliwe oprogramowanie, analiza ruchu sieciowego, protokół HTTP, klasyfikacja, rozpoznawanie otwartozbiorowe.

Abstract

Malware (malicious software) is a serious threat to the modern Internet. Criminals use it to send unsolicited messages, extort ransom by encrypting files, or steal bank login credentials. Malware utilizes popular network protocols for communication purposes, including the frequently used HyperText Transfer Protocol (HTTP). The goal of this dissertation is to demonstrate that malware-generated HTTP requests can be used for the identification and classification of malware families. For the purpose of this study, network traffic datasets covering 121 malware families and a set of popular benign applications were created. The conducted experimental evaluation was divided into three parts. The first part identified characteristic features of HTTP requests which allow distinguishing between malware and benign applications. Then, these features became the basis for the second part of the analysis, which resulted in the creation of a tool called *Hfinger*, that can to create unique representations of HTTP requests. These representations can be used to identify malware by distinguishing its families as well as its specific operations, e.g., attacks or downloading commands. Finally, the third part of the study focuses on the problem of classifying malware using machine learning algorithms, i.e., assigning the names of specific families to the analyzed network traffic. The problem is extended by the recognition of classes that were unknown during the training phase of the classifier, which is also called Open Set Recognition problem. During the experimental evaluation, two HTTP request representations were used: the first one generated by *Hfinger*, and the second one using n-gram analysis. To the author's best knowledge, this is the first work focused on the application of the Open Set Recognition approach to classify malware based on its HTTP network traffic.

Keywords: malicious software, malware, network traffic analysis, HTTP protocol, classification, Open Set Recognition.

Spis treści

Dorobek naukowy autora rozprawy **6**

1 Wstęp **9**

1.1 Tło badań 10

1.2 Motywacja i cel badań 12

1.3 Teza rozprawy 14

1.4 Przyjęte ograniczenia 16

1.5 Zakres badań oraz struktura rozprawy 18

2 Protokół HTTP oraz złośliwe oprogramowanie - podstawowe informacje **21**

2.1 Protokół HTTP 21

2.2 Złośliwe oprogramowanie 24

2.3 Użycie protokołu HTTP przez złośliwe oprogramowanie 26

3 Przegląd aktualnego stanu wiedzy i jego związek z problemem postawionym w rozprawie **29**

3.1 Charakteryzowanie ruchu sieciowego protokołu HTTP złośliwego oprogramowania 29

3.2 Identyfikacja żądań złośliwego oprogramowania 30

3.3 Wykrywanie złośliwego oprogramowania przy użyciu protokołu HTTP 34

3.4 Wykrywanie złośliwego oprogramowania w scenariuszu otwartozbiorowym . . 42

4 Analiza ruchu sieciowego protokołu HTTP złośliwego oprogramowania **49**

5 Identyfikacja żądań złośliwego oprogramowania **81**

6 Klasyfikacja rodzin złośliwego oprogramowania przy użyciu protokołu HTTP w scenariuszu otwartozbiorowym **111**

6.1 Wstęp 111

6.2 Problematyka rozpoznawania otwartozbiorowego 112

6.3 Opis badanych algorytmów uczenia maszynowego 114

6.3.1 Maszyny wektorów nośnych kalibrowane rozkładem Weibulla 114

6.3.2 Otwartozbiorowe maszyny wektorów nośnych 115

6.3.3 Maszyna wartości ekstremalnych . . . 117
6.4 Charakterystyka badań . . . 118
6.4.1 Dane badawcze . . . 118
6.4.2 Protokół badań . . . 121
6.4.3 Miary porównawcze . . . 124
6.4.4 Opis badań . . . 126
6.5 Wyniki badań . . . 131
6.5.1 Pełna reprezentacja narzędzia *Hfinger* . . . 131
6.5.2 Domyślna reprezentacja narzędzia *Hfinger* . . . 143
6.5.3 Porównanie wydajności reprezentacji narzędzia *Hfinger* oraz reprezentacji trigramowej . . . 156
6.6 Podsumowanie . . . 165
6.6.1 Wnioski . . . 165
6.6.2 Dyskusja . . . 167
6.6.3 Dalsze prace . . . 169

7 Podsumowanie **171**
7.1 Podsumowanie wyników prac . . . 171
7.2 Dyskusja . . . 175
7.3 Dalsze prace badawcze . . . 177

Literatura **179**

Spis rysunków **194**

Spis tabel **196**

Wykaz skrótów **198**

Dorobek naukowy autora rozprawy

Artykuły naukowe i rozdziały w książkach:

1. P. Białczak, W. Mazurczyk, Hfinger: Malware HTTP request fingerprinting, „Entropy", vol. 23, no. 5, 2021, ISSN: 1099-4300, doi: 10.3390/e23050507, https://www.mdpi.com/1099-4300/23/5/507

2. P. Białczak, W. Mazurczyk, Characterizing Anomalies in Malware-generated HTTP Traffic, "Security and Communication Networks", vol. 2020, pp. 1–26, Sep. 2020, https://doi.org/10.1155/2020/8848863

3. P. Białczak, Stan bezpieczeństwa certyfikatów X.509 najpopularniejszych polskich domen internetowych, XXXIII Krajowe Sympozjum Telekomunikacji i Teleinformatyki KSTiT 2017, Warszawa, wrzesień 2017, Materiały: Przegląd Telekomunikacyjny i Wiadomości Telekomunikacyjne, nr 8-9/2017, ss.923-928

4. P. Białczak, Sieci mobilne ad-hoc realizowane za pomocą interfejsu Bluetooth – praktyczne rozwiązania, w: Prace Seminarium Naukowego Instytutu Telekomunikacji Politechniki Warszawskiej. Tom 2 / Jakubiak Andrzej (red.), 2016, Oficyna Wydawnicza PW, ISBN 978-83-7814-581-3, ss. 11-20

5. P. Białczak, Badania nad nowymi metodami steganograficznymi - zarys problemu etycznego, w: Prace Seminarium Naukowego Instytutu Telekomunikacji Politechniki Warszawskiej. Tom 1 / Jakubiak Andrzej (red.), 2015, Oficyna Wydawnicza PW, ISBN 978-83-7814-394-9, ss. 87-100

6. P. Białczak, M. Dębska, Botnetowe domeny DGA: klasyfikacja metod wykrycia na przykładzie najważniejszych rozwiązań, XXXI Krajowe Sympozjum Telekomunikacji i Teleinformatyki KSTiT 2015, Kraków, wrzesień 2015, Materiały: Przegląd Telekomunikacyjny i Wiadomości Telekomunikacyjne, nr 8-9/2015

7. P. Białczak, W. Mazurczyk - Multipleksacja kanału ukrytego jako sposób niejawnej transmisji grupowej - XXX Krajowe Sympozjum Telekomunikacji i Teleinformatyki KSTiT 2014, Poznań,wrzesień 2014, Materiały: Przegląd Telekomunikacyjny i Wiadomości Telekomunikacyjne, nr 8-9/2014

8. P. Białczak, W. Mazurczyk, K. Szczypiorski - Sending Hidden Data via Google Suggest - In Proc. of: Third International Workshop on Network Steganography (IWNS 2011) co-located with The 2011 International Conference on Telecommunication Systems, Modeling and Analysis (ICTSM2011), ISBN 978-0-9820958-4-3, pp. 121-131, Prague, Czech Republic, 26-28.05.2011

9. P. Białczak, W. Mazurczyk, K. Szczypiorski - Ukrywanie informacji w Google Suggest – XXVII Krajowe Sympozjum Telekomunikacji i Teleinformatyki KSTiT 2011, Łódz, 14-16 września 2011, Materiały: Przegląd Telekomunikacyjny i Wiadomości Telekomunikacyjne, nr 8-9/2011, str. 789-794

Raporty techniczne i inne:

1. P. Białczak, P. Pawliński, K. Rydz, R. Mattioli, Proactive detection – Measures and information sources, ENISA technical report, maj 2020, `https://www.enisa.europa.eu/publications/proactive-detection-measures-and-information-sources`, dostęp on-line: 23.03.2022 r.

2. W. Sypniewski, P. Białczak, Network traffic periodicty analysis of dark IP address space, technical report, CERT Polska, 01/08/2016, `https://www.cert.pl/wp-content/uploads/2016/08/dark_space_report.pdf`, dostęp on-line: 09.05.2017 r.

Projekty badawcze:

1. SISSDEN – Secure Information Sharing Sensor Delivery Event Network, międzynarodowy projekt europejski w ramach programu H2020, rola w projekcie: junior researcher, inżynier oprogramowania

2. microMole – Sewage monitoring system for tracking synthetic drug laboratories, międzynarodowy projekt europejski w ramach programu H2020, rola w projekcie: konstruktor

Prezentacje konferencyjne:

1. Śledzenie złośliwego oprogramowania przy wykorzystaniu protokołu HTTP, XXIV Konferencja na temat bezpieczeństwa teleinformatycznego Secure 2020, Warszawa, październik 2020

2. Hunting malware using its fingerprints, x33fcon Security Conference, Gdynia, wrzesień 2020

3. Blokowanie niebezpiecznych stron przez wykorzystanie listy ostrzeżeń CERT Polska, SysOps/DevOps Online MeetUp – Lightning Talk, 27.08.2020

4. Tracking botnets with Long Term Sandboxing - Botconf 2019, The botnet fighting conference, Bordeaux, grudzień 2019 (razem z A. Korczakiem)

5. LTS – czyli jak sandboksy długoterminowe pomagają w śledzeniu botnetów, XXIII Konferencja na temat bezpieczeństwa teleinformatycznego Secure 2019, październik 2019 (razem z A. Korczakiem)

6. Leaving no stone unturned – in search of HTTP malware distinctive features, Botconf 2018, The botnet fighting conference, Toulouse, grudzień 2018

7. How long term malware sandboxing can help fight electronic crime?, APWG.EU EU Symposium on Electronic Crime Research, 17-19 września 2018, Kraków

8. Dwa światy złośliwych żądań HTTP, XXI Konferencja na temat bezpieczeństwa teleinformatycznego Secure 2017, październik 2017

9. Stan bezpieczeństwa certyfikatów X.509 najpopularniejszych polskich domen internetowych, XXXIII Krajowe Sympozjum Telekomunikacji i Teleinformatyki KSTiT 2017, Warszawa, wrzesień 2017

10. Dlaczego sygnatury już nie wystarczają: o wykrywaniu domen generowanych algorytmicznie w botnetach, XIX Konferencja na temat bezpieczeństwa teleinformatycznego Secure 2015 (razem z M. Dębską)

11. Multipleksacja kanału ukrytego jako sposób niejawnej transmisji grupowej - XXX Krajowe Sympozjum Telekomunikacji i Teleinformatyki KSTiT 2014, Poznań, wrzesień 2014

12. Ukrywanie informacji w Google Suggest - XXVII Krajowe Sympozjum Telekomunikacji i Teleinformatyki KSTiT 2011, Łódz, 14-16 września 2011

1 Wstęp

Liczba incydentów bezpieczeństwa komputerowego ostatnimi laty ciągle wzrasta. Według CERT Polska, pełniącego rolę zespołu CSIRT (Computer Security Incident Response Team) poziomu krajowego w Polsce, liczba tego typów incydentów w 2021 roku wzrosła o 182% względem 2020 roku [61]. Mimo że oszustwa komputerowe stanowiły większość incydentów w Polsce (86,40%), to drugą najliczniejszą grupę stanowiły te powiązane ze złośliwym oprogramowaniem (9,66%). Także Agencja Unii Europejskiej ds. Cyberbezpieczeństwa (ENISA) w swoim raporcie dotyczącym krajobrazu zagrożeń cyberbezpieczeństwa [26], określa złośliwe oprogramowanie jako jedno z głównych zagrożeń, mimo zmniejszania się jego znaczenia od 2020 roku.

Terminem złośliwego oprogramowania (malicious software - malware) określa się zbiór aplikacji mających na celu wyrządzenie szkód lub zniszczenie komputerów, lub systemów komputerowych[1]. Wykorzystywane jest ono do różnych celów: wysyłania niechcianych wiadomości, wykradania danych logowania do banków, atakowania serwerów czy wymuszania okupu przez szyfrowanie plików. Autorzy złośliwego oprogramowania mogą być postrzegani jako hakerzy, czyli osoby ekspercko wykorzystujące systemy komputerowe, także do łamania zabezpieczeń. S. Chng i in. [21] wyróżniają 13 typów hakerów, których może motywować: ciekawość, aspekty finansowe, rozgłos, zemsta, zabawa, ideologia lub impulsy seksualne. Wśród nich można wyróżnić np. cyberprzestępców, hacktywistów, wandali internetowych czy przedstawicieli organów państwowych.

Działanie złośliwego oprogramowania może powodować różne efekty[2]: zaburzyć funkcjonowanie systemów komputerowych, np. przez skasowanie ważnych plików lub całkowite wykorzystanie zasobów sprzętowych, wywołać fizyczne awarie sprzętu, a także spowodować utratę danych lub ich wykradnięcie. Każdy z tych efektów powoduje straty finansowe, zarówno dla zwykłego użytkownika internetu, jak i dla dużych firm, choć w ich przypadku ze względu na efekt skali, straty mogą być znacząco większe. Według raportu IBM Security [91], średni koszt nieuprawnionego dostępu do danych (data breach) ze względu na pierwotny wektor ataku może sięgać ok. 5 milionów dolarów - w przypadku przejęcia biznesowego konta pocztowego, lub 4,37 miliona - w przypadku przejęcia danych logowania. Oba sposoby ataku mogą zostać wy-

[1] `https://www.cisco.com/c/en/us/products/security/advanced-malware-protection/what-is-malware.html`. Dostęp: 27.03.2022 r.

[2] `https://encyclopedia.kaspersky.com/knowledge/damage-caused-by-malware/`. Dostęp: 27.03.2022 r.

konane przy użyciu złośliwego oprogramowania. Dosyć poważnym zagrożeniem jest także tzw. ransomware, czyli oprogramowanie szyfrujące dane w celu wymuszenia okupu. Według raportu Cybereason [25] 66% firm zaatakowanych przez ten typ złośliwego oprogramowania doznało znaczących strat finansowych, a średnia kwota okupu rośnie z każdym rokiem, np. w 2018 r. było to 6 tys. dolarów, a w 2020 już 178 tys.

Powyższe przykłady wskazują jak istotnym zagrożeniem dla użytkowników internetu jest złośliwe oprogramowanie. Walkę z tym zagrożeniem ułatwia jego efektywna identyfikacja oraz wykrywanie przez analizę zachowania. Właśnie temu zagadnieniu poświęcona jest niniejsza rozprawa.

1.1 Tło badań

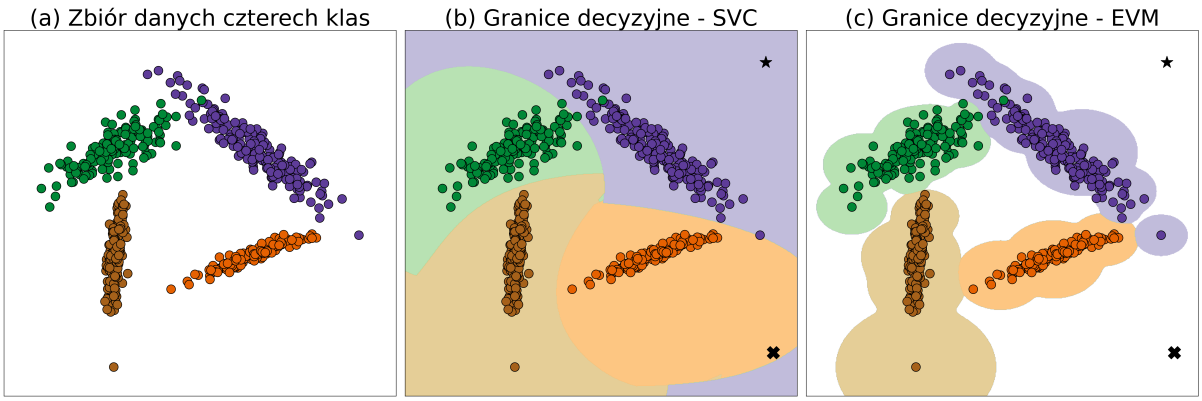
Wykorzystanie analizy zachowania umożliwia lepsze poznanie złośliwego oprogramowania, identyfikację jego charakterystycznych cech, a docelowo także wykrycie. Pod pojęciem zachowania mieszczą się np. wywołania bibliotek systemu operacyjnego lub zmiany w strukturze plików, ale także przesyłany ruch sieciowy. Generowany jest on przez wykorzystanie Internetu w celu m.in. pobrania komend z serwerów zarządzających (Command and Control - C&C), wykonania ataków lub wysłania zebranych danych. Do komunikacji złośliwe oprogramowanie wykorzystuje różne popularne protokoły w tym protokół HTTP (HyperText Transfer Protocol), którego wersja 1.1 została zdefiniowana pierwotnie w RFC 2616 [35]. HTTP działa w architekturze klient-serwer, w której klient wysyła żądania o pewne zasoby, które serwer udostępnia. Dane wymieniane w wersji 1.1 tego protokołu mają reprezentację tekstową, łatwą do odczytania przez człowieka.

Protokół HTTP może być wykorzystany do wykrywania malware'u np. przez poszukiwanie pewnej wyróżniającej cechy, wskazującej na złośliwość ruchu sieciowego. Poszukiwanie takich cech i tworzenie na ich bazie reguł detekcji może nie być efektywne czasowo oraz skalowalne do większej liczby rodzin złośliwego oprogramowania. W tym przypadku możliwym rozwiązaniem jest użycie metod uczenia maszynowego, które pozwalają wykorzystać zależności w danych, które nie zawsze są widoczne dla człowieka. Metody te stosuje się do zadań wykrywania i klasyfikacji. W niniejszej rozprawie wykrywanie, inaczej detekcja, definiowane będzie jako określenie czy badane zachowanie jest złośliwe czy nie. Z kolei klasyfikacja definiowana będzie jako rozpoznanie i przyporządkowanie analizowanego zachowania do konkretnej rodziny złośliwego oprogramowania lub aplikacji niezłośliwej.

W standardowym podejściu do klasyfikacji, w czasie predykcji, tzn. procesie przyporządkowania klasy do zebranych danych, występują jedynie próbki danych *klas znanych*, czyli klas obecnych w trakcie treningu klasyfikatora. Przykładem może być system rozpoznawania zwierząt na obrazkach. Do treningu wykorzystywane są pliki graficzne z wizerunkiem zwierząt, które zostały uprzednio ponazywane. Dzięki temu, w trakcie predykcji przy napotkaniu nowego obrazka zostanie mu przypisana odpowiednia nazwa spośród tych poznanych w trakcie treningu. Niemniej, w systemie używanym w rzeczywistych warunkach, klasyfikacji mogą zostać poddane pliki z wizerunkiem innego typu obiektu, np. krzesła, lub dotychczas nieznanego, niedawno odkrytego zwierzęcia. W takiej sytuacji standardowy klasyfikator przypisze błędną nazwę spośród tych wyuczonych w treningu, nie informując o napotkaniu reprezentanta dotychczas niewidzianej klasy. Dlatego w celu lepszego odzwierciedlenia rzeczywistych scenariuszy użycia, problem klasyfikacji można rozszerzyć o wykrywanie klas, które były nieznane w trakcie treningu klasyfikatora. Taki scenariusz działania jest nazywany klasyfikacją otwartozbiorową (open set classification) lub rozpoznawaniem otwartozbiorowym (open set recognition) [42]. Określenie *otwartozbiorowy* wskazuje na możliwość wystąpienia w trakcie predykcji klasyfikatora próbek *klas nieznanych*, tzn. takich klas, które nie znajdowały się w zbiorze danych treningowych danego klasyfikatora. Jest to sytuacja odmienna od scenariusza *zamkniętozbiorowego* (closed set), znanego ze standardowych podejść do klasyfikacji.

Problem otwartozbiorowy został zaprezentowany na rys. 1.1. Czteroklasowy zbiór danych zaprezentowany w części (a) został częściowo użyty do treningu dwóch klasyfikatorów. Pierwszy klasyfikator SVC (Support Vector Classification) bazuje na algorytmie maszyn wektorów nośnych i jest klasyfikatorem zamkniętozbiorowym. Drugi klasyfikator jest otwartozbiorowy i wykorzystuje maszyny wektorów ekstremalnych (Extreme Value Machine - EVM) zaproponowane przez E. M. Rudda i in. w pracy [88]. Rys. 1.1 w częściach (b) i (c) prezentuje granice decyzyjne w kolejności klasyfikatorów SVC i EVM. Granice te określają, które punkty w przestrzeni danych wejściowych zostaną przyporządkowane do konkretnej rodziny. Granice decyzyjne klasyfikatora SVC są nieograniczone i wypełniają całą przestrzeń danych. Z tego powodu punkty danych klas nieznanych symbolizowane krzyżykiem i gwiazdką, mimo że znajdują się w znacznej odległości od punktów treningowych, zostaną zaklasyfikowane do jednej z klas znanych w trakcie treningu klasyfikatora. Granice decyzyjne klasyfikatora EVM zaprezentowane w części (c) są widocznie ograniczone. Występuje tu obszar bez koloru wypełnienia, oznaczający, że punkty danych mu odpowiadające nie należą do żadnej ze znanych klas. W

związku z tym, punkty danych oznaczone krzyżykiem i gwiazdką zostaną zaklasyfikowane jako należące do klasy nieznanej.



Rysunek 1.1: Przykład problemu otwartozbiorowego: (a) zbiór danych czterech klas; (b) granice decyzyjne standardowego klasyfikatora zamkniętozbiorowego SVC; (c) granice decyzyjne klasyfikatora otwartozbiorowego maszyny wektorów ekstremalnych EVM. Punkty danych klas nieznanych są oznaczone gwiazdką i krzyżykiem. Kształt granic decyzyjnych klasyfikatora SVC powoduje przypisanie klas nieznanych do dwóch klas znanych. Klasyfikator EVM poprawnie uznaje je za punkty klasy nieznanej.

Scenariusz wystąpienia klas nieznanych jest możliwy także w klasyfikatorze złośliwego oprogramowania, kiedy to w trakcie predykcji mogą wystąpić próbki wcześniej niewidzianych rodzin, np. takich, które zostały niedawno stworzone. Zatem podejście otwartozbiorowe umożliwia lepsze modelowanie rzeczywistych problemów przed którymi stoją systemy analizy malware'u.

1.2 Motywacja i cel badań

Protokół HTTP, jako jeden z najpopularniejszych protokołów wykorzystywanych przez złośliwe oprogramowanie (por. S. Miller i in. [73]), umożliwia analizę jego zachowania, odkrycie specyficznych cech, a także jego detekcję i klasyfikację. Istnieją różne podejścia badawcze do wykorzystania tego protokołu do wymienionych celów (por. np. D. Acarali i in. [1]). Część z podejść analizuje ruch sieciowy protokołu HTTP malware'u w celu odkrycia jego cech charakterystycznych – np. C. Rossow i in. [87], M. C. Calzarossa i in. [19] czy R. Montoro [76]. Niestety, liczba analizowanych w ten sposób cech jest ograniczona, a zależności między nimi nie są wystarczająco zbadane. W innych podejściach tworzone są modele ruchu sieciowego bazujące

na omawianym protokole, które umożliwiają identyfikację złośliwego oprogramowania – jak np. R. Bortolameotti i in. w [15], [16] czy Z. Tang i in. w [100]. W podejściach tych konieczne jest przeprowadzenie fazy tworzenia modelu monitorowanej sieci. Konieczne jest także użycie zestawów żądań lub żądań i odpowiedzi HTTP. Dodatkowo niewykorzystywane są wszystkie informacje dostępne w przesyłanych wiadomościach. Wreszcie istnieją prace traktujące o detekcji lub klasyfikacji złośliwego oprogramowania przy użyciu protokołu HTTP, jak np. R. Perdisci i in. w [80], N. Kheir w [57], S. Mizuno i in. w [74] czy J. Xie i in. w [110]. Niestety, tylko niektóre rozwiązania tego typu umożliwiają klasyfikację rodzin malware'u. A informacje przesyłane w wiadomościach protokołu nie zawsze są w pełni wykorzystywane.

Kluczowym problemem dotychczasowych badań nad detekcją i klasyfikacją złośliwego oprogramowania przy użyciu protokołu HTTP jest brak możliwości pracy w scenariuszu otwartozbiorowym. Oznacza to, że, według wiedzy autora, żadne z dotychczas opublikowanych źródeł nie umożliwia klasyfikacji rodzin malware'u oraz nie wykrywa przy tym obecności klas nieznanych, tzn. niewidzianych wcześniej rodzin.

Celem poniższej rozprawy jest próba odpowiedzi na powyższe problemy napotykane w badaniach analizujących złośliwe oprogramowanie przy użyciu ruchu sieciowego protokołu HTTP. Dokonywane jest to poprzez przeprowadzenie następujących kroków:

1. Przeprowadzenie dogłębnej i systematycznej analizy różnych cech żądań HTTP na bazie dużej liczby rodzin złośliwego oprogramowania. Analiza ta została zaprezentowana w rozdziale 4.

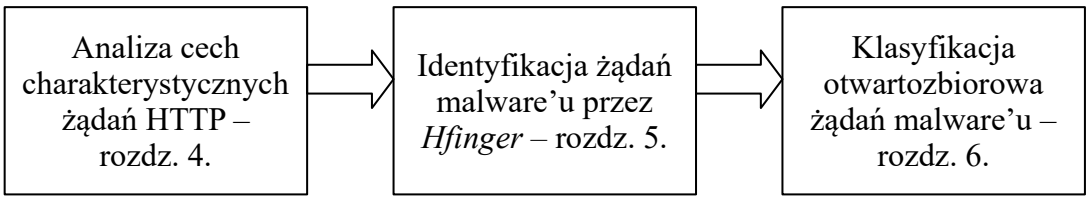
2. Stworzenie narzędzia *Hfinger*, tworzącego reprezentacje pojedynczych żądań HTTP, co umożliwia identyfikację złośliwego oprogramowania. Przypisanie jednoznacznych i unikalnych reprezentacji do żądań wygenerowanych przez próbkę złośliwego oprogramowania pozwala stworzyć swoisty odcisk palca, w pewien sposób modelujący zachowanie badanej próbki. Dzięki temu staje się możliwe rozróżnianie różnych rodzin malware'u, tym samym zapewniając ich identyfikację[3]. Opracowane narzędzie wykorzystuje szeroki zakres danych dostępnych w różnych częściach żądania, bazując na plikach pcap z zapisanym ruchem sieciowym i nie wymagając przy tym tworzenia modeli zachowania aplikacji obecnych w monitorowanej sieci. Narzędzie *Hfinger* jest publicznie dostępne pod adresem `https://github.com/CERT-Polska/hfinger`. Szczegóły przeprowadzonych

[3]Identyfikacja nie może być rozumiana jako przypisanie nazwy konkretnej rodziny – tym zadaniem zajmuje się klasyfikacja.

badań zostały przedstawione w rozdziale 5.

3. Porównanie skuteczności klasyfikacji różnych algorytmów uczenia maszynowego, wykorzystujących reprezentację żądań HTTP stworzoną przez system *Hfinger*, umożliwiając przy tym działanie zgodnie ze scenariuszem otwartozbiorowym. W tym kroku przebadano różne algorytmy klasyfikacji otwartozbiorowej oraz tradycyjne algorytmy uczenia maszynowego. Dokonano także porównania wydajności klasyfikacji między reprezentacjami żądań HTTP bazującymi na narzędziu *Hfinger* oraz reprezentacją bazującą na analizie n-gramowej (por. D. Jurafsky i in. [55]). Opis przeprowadzonych badań został zaprezentowany w rozdziale 6. Według wiedzy autora jest to pierwsza publiczna analiza systemów klasyfikacji otwartozbiorowej wykorzystujących ruch sieciowy protokołu HTTP.

Rys. 1.2 prezentuje schemat zależności wyników kolejnych kroków badań od siebie.



Rysunek 1.2: Zależność wyników kolejnych kroków badań od siebie.

Cele rozprawy zostały formalnie przedstawione w kolejnym podrozdziale.

1.3 Teza rozprawy

Teza niniejszej rozprawy została sformułowana w następujący sposób:

Wykorzystanie żądań protokołu HTTP umożliwia identyfikację złośliwego oprogramowania poprzez stworzenie unikalnej reprezentacji jego ruchu sieciowego oraz skuteczne rozpoznanie rodziny złośliwego oprogramowania, a także obecności przedstawicieli rodzin dotychczas nieznanych.

W celu udowodnienia powyższej tezy podjęto decyzję o podzieleniu jej na części składowe - podtezy, które zostaną udowodnione w dalszych rozdziałach. Udowodnienie tych części, pozwoli następnie udowodnić także tezę główną. Wspomniane podtezy można sformułować w postaci:

1. **Ruch sieciowy żądań protokołu HTTP złośliwego oprogramowania zawiera cechy charakterystyczne, które umożliwiają jego odróżnienie od ruchu sieciowego aplikacji niezłośliwych.**

   W podtezie tej zakłada się, że istnieją pewne cechy w żądaniach HTTP generowanych przez złośliwe oprogramowanie, które nie występują w żądaniach aplikacji niezłośliwych lub są bardzo rzadkie. Na bazie częstości wystąpienia tych cech możliwe jest odróżnienie tych dwóch typów oprogramowania.

   Sposób udowodnienia: Ruch sieciowy żądań HTTP złośliwego oprogramowania oraz aplikacji niezłośliwych analizuje się pod kątem częstości występowania zestawu zdefiniowanych cech. Częste występowanie pewnych cech w ruchu sieciowym złośliwego oprogramowania przy braku lub małej częstości występowania w ruchu aplikacji niezłośliwych udowodni postawioną składową tezy. Badane cechy zostały sformułowane na bazie analiz innych badaczy oraz doświadczenia autora przy analizie malware'u. Udowodnieniem tej tezy zajmuje się rozdział 4.

2. **Możliwe jest stworzenie unikalnej reprezentacji żądań protokołu HTTP, która umożliwia identyfikację złośliwego oprogramowania.**

   W podtezie tej zakłada się, że jest możliwe stworzenie transformacji informacji zawartych w żądaniu HTTP do postaci, która będzie w sposób jednoznaczny identyfikowała żądania złośliwego oprogramowania. Pomysł stworzenia takiego sposobu reprezentacji bazuje częściowo na tezie 1. oraz cechach badanych przy jej udowodnieniu.

   Sposób udowodnienia: Zaprojektowanie reprezentacji żądań HTTP i wykorzystanie jej do analizy ruchu sieciowego złośliwego oprogramowania oraz aplikacji niezłośliwych. Badana tu jest efektywność reprezentacji rozumiana jako unikalność stworzonych reprezentacji, a także stopień możliwości grupowania podobnych żądań. Unikalność bazuje na maksymalizacji odróżnialności rodzin złośliwego oprogramowania od siebie. Umożliwia to ich identyfikację przez jednoznaczną reprezentację zachowania, na które składają się żądania HTTP. Udowodnienie tej składowej tezy odbędzie się przez wskazanie wyższej efektywności zaprojektowanej reprezentacji w stosunku do innych stosowanych w popularnych narzędziach analizy. Opis związanych z tym prac zawarto w rozdziale 5.

3. **Wykorzystanie odpowiednio stworzonej reprezentacji żądań HTTP umożliwia skuteczne rozpoznanie rodzin złośliwego oprogramowania, w tym także istnienia klas**

dotychczas nieznanych.

W powyższej podtezie zakłada się, że reprezentacja żądań HTTP zaprojektowana do udowodnienia składowej tezy 2. może być wykorzystana do klasyfikacji złośliwego oprogramowania w scenariuszu otwartozbiorowym.

Sposób udowodnienia: Zaprojektowana reprezentacja żądań wykorzystywana jest do stworzenia cech wejściowych dla różnych algorytmów uczenia maszynowego. Dzięki wykorzystaniu uprzednio zebranego ruchu sieciowego złośliwego oprogramowania oraz aplikacji niezłośliwych możliwe staje się wykonanie badań skuteczności klasyfikacji otwartozbiorowej tych algorytmów. Przez skuteczność klasyfikacji należy tu rozumieć poziomy zdefiniowanych miar poprawności przypisania nazwy rodziny, a także czas wykonania obliczeń. Przeprowadzone badania umożliwiają wybranie najlepszych algorytmów. Teza staje się udowodniona, gdy poziomy skuteczności klasyfikacji są dobre, a także gdy są porównywalne lub lepsze na tle innego sposobu reprezentacji żądań. W rozdziale 6. znajduje się opis badań udowadniających tę składową tezy.

1.4 Przyjęte ograniczenia

W badaniach przedstawionych w poniższej rozprawie przyjęto kilka ograniczeń. Przede wszystkim w analizie ograniczono się do wykorzystania tylko protokołu HTTP, a w nim tylko żądań. Wybór tylko tego protokołu podyktowany jest jego dużą popularnością użycia przez złośliwe oprogramowanie (por. S. Miller i in. [73]). Z kolei skupienie się jedynie na żądaniach bierze się z faktu, że odpowiedzi serwerów nie zawsze są poprawne lub w ogóle nie są dostępne. Do takiej sytuacji może dojść, np. gdy infrastruktura przestępców została przejęta przez badaczy bezpieczeństwa (np. wyniku tzw. sinkholingu) przez co zwracane odpowiedzi mogą nie być takie same, jak gdyby były wysyłane przez oryginalny serwer.

Następnie, badania skupiają się na wykorzystaniu protokołu HTTP w wersji co najwyżej 1.1, choć istnieją także wersje 2. i 3. (więcej można o nich przeczytać w rozdziale 2.). Wprowadzają one skuteczniejsze sposoby przekazywania wiadomości, w związku z czym dane nie są już przesyłane w nich w formie tekstowej jak w wersji 1.1. Niemniej, wiadomości przed zakodowaniem są w nich tworzone zgodnie z semantyką starszej wersji protokołu. Z tego powodu można na nie rozszerzyć prezentowane analizy.

Z kolei w rozdziale 6. zaprezentowano badania nad klasyfikacją złośliwego oprogramowania wykorzystujące metody uczenia maszynowego. W badaniach tych nie wykorzystywano sieci

neuronowych z kilku powodów. Po pierwsze obecne mechanizmy otwartozbiorowe dla sieci neuronowych są dopasowane do innych reprezentacji danych niż te generowane przez narzędzie *Hfinger*. Przykładowo, warstwa OpenMax zaprezentowana przez A. Bendale i in. w [9] w oryginale została dostosowana do analizy obrazów w splotowej sieci neuronowej. Dostosowanie jej do reprezentacji tworzonej przez *Hfinger* jest możliwe, ale należałoby się zastanowić nad sensem takiego podejścia, jeśli potencjalnie inne sposoby reprezentacji żądań HTTP byłyby wydajniejsze. Tu uwidacznia się drugi problem, w którym właśnie należałoby znaleźć odpowiednią reprezentację żądań, lepszą z punktu widzenia zastosowania w sieciach neuronowych. Jednym z podejść mogłaby być bezpośrednia zamiana danych żądań HTTP na obrazy w skali szarości, co umożliwiałoby wykorzystanie sieci specjalizujących się w analizie obrazów. Innym podejściem mogłaby być analiza danych żądań bezpośrednio w formie tekstowej, do czego można by użyć np. wyspecjalizowane rekurencyjne sieci neuronowe. Wreszcie, wykorzystanie nowych reprezentacji żądań, a także tworzenie wyspecjalizowanych sieci neuronowych umożliwiających działanie w scenariuszu otwartozbiorowym, powoduje stosunkowo duży narzut czasowo-badawczy w porównaniu do wykorzystania gotowych algorytmów otwartozbiorowych, wpisujących się w klasyczne metody uczenia maszynowego. Jako że poniższa rozprawa, według wiedzy jej autora, jest pierwszym podejściem do problemu klasyfikacji złośliwego oprogramowania przy użyciu żądań HTTP w scenariuszu otwartozbiorowym, jednym z priorytetów badań było potwierdzenie skuteczności tego scenariusza pod względem klasyfikacji. Wykorzystanie gotowych algorytmów, wymagających mniej pracy badawczo-rozwojowej niż sieci neuronowe, umożliwiło, jak się okaże w rozdziale 6., potwierdzenie tej skuteczności przy zużyciu mniejszych zasobów.

Dosyć ważnym ograniczeniem badań jest skupienie się na złośliwym oprogramowaniu atakującym systemy operacyjne z rodziny Windows. Mimo popularności innych systemów operacyjnych, w tym mobilnych takich jak Android, nadal najwięcej złośliwego oprogramowania tworzone jest na platformę Windows[4]. Z tego powodu to właśnie na niej skupiły się prace badawcze. Niemniej, podejścia zaprezentowane w niniejszej rozprawie można potencjalnie rozszerzyć także na inne systemy operacyjne.

[4] `https://portal.av-atlas.org/malware`, dostęp: 23.03.2022 r.

1.5 Zakres badań oraz struktura rozprawy

Na niniejszą rozprawę oraz badania, które doprowadziły do jej powstania składają się następujące czynności:

- dokonano przeglądu aktualnego stanu wiedzy dotyczącego analizy ruchu sieciowego protokołu HTTP złośliwego oprogramowania, w tym identyfikacji żądań złośliwego oprogramowania i ich detekcji, a także wykrywania złośliwego oprogramowania w scenariuszu otwartozbiorowym,
- przeprowadzono ocenę bieżącego stanu wiedzy dotyczącego analizy ruchu sieciowego protokołu HTTP złośliwego oprogramowania, wskazano główne problemy aktualnych rozwiązań,
- stworzono zbiory danych ruchu sieciowego HTTP złośliwego oprogramowania oraz aplikacji niezłośliwych, które posłużyły do przeprowadzenia prezentowanych w rozprawie badań eksperymentalnych,
- wykonano przegląd cech żądań HTTP wykorzystywanych jako anomalie lub wskaźniki złośliwości,
- zaproponowano nowe cechy, które potencjalnie można wykorzystać do odróżnienia ruchu sieciowego złośliwego oprogramowania,
- przeanalizowano i zidentyfikowano charakterystyczne cechy żądań HTTP złośliwego oprogramowania oraz ich wzajemne oddziaływanie,
- stworzono listę charakterystycznych cech żądań HTTP złośliwego oprogramowania oraz sposoby ich praktycznego użycia,
- zaprojektowano i skonstruowano narzędzie *Hfinger* umożliwiające identyfikację żądań HTTP złośliwego oprogramowania,
- przeprowadzono przegląd oraz analizę znanych i popularnych narzędzi do identyfikacji żądań HTTP,
- porównano efektywność identyfikacji żądań HTTP złośliwego oprogramowania narzędzia *Hfinger* w stosunku do innych popularnych rozwiązań do identyfikacji żądań HTTP,

- przystosowano do jednorodnego interfejsu programistycznego API popularne algorytmów otwartozbiorowych,
- przeprowadzano analizę algorytmów otwartozbiorowych pod kątem wydajności klasyfikacji i czasu koniecznego do wykonania obliczeń na tle tradycyjnych algorytmów uczenia maszynowego,
- przeprowadzano analizę porównawczą reprezentacji żądań HTTP bazujących na narzędziu *Hfinger* oraz analizie n-gramowej,
- oceniono przydatność badanych algorytmów oraz reprezentacji żądań do klasyfikacji złośliwego oprogramowania w scenariuszu otwartozbiorowym.

W toku prac badawczych, których efektem jest niniejsza rozprawa, powstały następujące artykuły naukowe:

- Piotr Białczak i Wojciech Mazurczyk, "Characterizing anomalies in malware-generated HTTP traffic", *Security and Communication Networks*, vol. 2020, pp. 1–26, Sierpień 2020, DOI: 10.1155/2020/8848863 (*IF = 1,791*)
- Piotr Białczak i Wojciech Mazurczyk, "Hfinger: Malware HTTP request fingerprinting", *Entropy*, vol. 23, no. 5, 2021, ISSN: 1099-4300. DOI: 10.3390/e23050507 (*IF = 2,524*)

Rozprawa została podzielona na 7 rozdziałów. W jej dalszej części znajduje się:

- W rozdziale 2. umieszczono podstawowe informacje o protokole HTTP: sposobie działania, strukturze wiadomości oraz sposobie reprezentacji danych. Umieszczono tam także informacje o złośliwym oprogramowaniu: podział na typy, opis komunikacji, a także sposoby analizy i detekcji.
- W rozdziale 3. dokonano przeglądu aktualnego stanu wiedzy dotyczącego wykorzystania protokołu HTTP do identyfikacji i klasyfikacji złośliwego oprogramowania. Dokonano omówienia i analizy krytycznej najważniejszych prac, głównych zastrzeżeń co do nich oraz związku z problematyką rozprawy.
- W rozdziale 4. przedstawiono analizę ruchu sieciowego protokołu HTTP złośliwego oprogramowania. Opisano użyte zbiory danych, analizowane cechy oraz sposób badań. Wskazano także najważniejsze cechy wyróżniające żądania HTTP złośliwego oprogramowania.

- W rozdziale 5. zaprezentowano badania nad identyfikacją żądań HTTP złośliwego oprogramowania. Opisano tam także sposób tworzenia reprezentacji żądań w autorskim narzędziu *Hfinger* oraz dokonano analizy jego efektywności w porównaniu do innych podobnych rozwiązań.

- Rozdział 6. opisuje badania klasyfikacji złośliwego oprogramowania w scenariuszu otwartozbiorowym wykorzystujące reprezentacje żądań HTTP tworzone przez narzędzie *Hfinger*. Zaprezentowano wykorzystywane algorytmy oraz sposób ich wykorzystania. Porównano wyniki efektywności klasyfikacji dwóch reprezentacji narzędzia *Hfinger*, a także popularnego podejścia wykorzystującego reprezentację n-gramową.

- W ostatnim rozdziale 7. znajduje się podsumowanie rozprawy. Streszcza ono najważniejsze wnioski oraz obserwacje, a także analizuje wyniki pod kątem udowodnienia tezy rozprawy. Omówione zostały tam też zauważone problemy prezentowanych badań, a także potencjalne obszary dalszych analiz.

2 Protokół HTTP oraz złośliwe oprogramowanie - podstawowe informacje

W poniższym rozdziale zamieszczono podstawowe informacje dotyczące protokołu HTTP oraz złośliwego oprogramowania, wymagane do lepszego zrozumienia tematyki prezentowanej w dalszej części rozprawy.

2.1 Protokół HTTP

Protokół HTTP w wersji 1.1 bazuje na architekturze klient-serwer, w której klient wysyła żądanie, a serwer na nie odpowiada. Oryginalnie wersja 1.1 protokołu została zdefiniowana w RFC 2616 [35]. Niemniej wprowadzono zmiany w jego definicji w RFC 7230-7235 [36]–[41]. Istnieją dwie wcześniejsze wersje protokołu: 0.9 and 1.0, ta ostatnia zdefiniowana w RFC 1945 [10].

W 2015 r. opublikowano dokument RFC 7540 [8] wprowadzający wersję 2. protokołu HTTP, natomiast w momencie pisania tej rozprawy (początek 2022r.) trwają prace nad standaryzacją wersji 3. protokołu [13]. Zmiany wprowadzone w wersjach 2. i 3. polegają głównie na przejściu od reprezentacji tekstowej do binarnej oraz wykorzystaniu wydajniejszych metod warstwy transportowej, np. przez użycie mechanizmów umożliwiających multipleksację danych lub użycie protokołu QUIC zamiast TCP [13]. Co ważne, w obydwu wersjach semantyka protokołu HTTP w wersji 1.1 pozostaje zachowana. W praktyce oznacza to, że żądania nadal są tworzone tekstowo według semantyki wersji 1.1 protokołu, ale w celu przesłania danych są one przekształcane do reprezentacji binarnej. Dalszy tekst w niniejszym rozdziale dotyczy wersji 1.1 protokołu.

Przykład żądania HTTP umieszczono w listingu 1.

Listing 1: Listing przykładowego żądania HTTP.

```
GET / HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:63.0) Gecko/20100101 Firefox/63.0
Accept: text/html,application/xhtml+xml, application/xml;q=0.9,*/*;q=0.8
Accept-Language: pl,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Pierwsza linia żądania rozpoczyna się od podania metody żądania, która opisuje jaki rodzaj

akcji ma zostać wykonany na żądanym zasobie. Zasób w postaci URI (Uniform Resource Identifier) znajduje się zaraz za metodą. W Listingu 1 metoda „GET" oznacza żądanie zasobu znajdującego w głównym katalogu - „/". Po definicji metody i zasobu znajduje się informacja o wersji protokołu - w tym przypadku „HTTP/1.1". Na końcu linii umieszczono znacznik powrotu karetki i nowej linii, oznaczany jako CRLF (Carriage Return Line Feed). Znacznik ten oddziela kolejne linie żądania, co ułatwia ich parsowanie. Kolejne linie żądania zawierają pola nagłówków, każde z nazwą pola i wartością pola, oddzielone dwukropkiem „:". W nazwach pól nie jest rozróżniana wielkość liter, a wartości pól zwykle składają się z drukowalnych znaków US-ASCII. Większość implementacji klientów HTTP w praktyce postępuje zgodnie z tą zasadą i używa w polach nagłówków tylko takich znaków. Dodatkowo, RFC 7230 uznało używanie znaków nieznajdujących w tablicy US-ASCII w wartościach nagłówków za przestarzałe. Po definicjach nagłówków w żądaniu umieszczany jest dodatkowy znacznik CRLF. Jeżeli w żądaniu mają zostać wysłane dane (payload), to są one umieszczane po ostatnim znaczniku CRLF. Pole danych może także się kończyć tym znacznikiem.

Poniżej zaprezentowano metody żądań zdefiniowane w RFC 7231 [39]. Każda z metod definiuje jakiej akcji dotyczy żądanie na wskazanym zasobie:

- *GET* – podstawowa metoda żądania dostępu do zasobu. Zwykle nie zawiera pola z danymi, ale jego obecność nie jest zabroniona.
- *HEAD* – metoda podobna do *GET*, z tym wyjątkiem, że serwer nie może odesłać żadnych danych w odpowiedzi - oprócz nagłówków,
- *POST* – metoda wskazująca by serwer przetworzył dane zawarte w polu danych żądania,
- *PUT* – metoda używana do stworzenia lub podmienienia stanu żądanego zasobu na ten umieszczony w polu danych żądania,
- *DELETE* – metoda używana do usunięcia powiązania między zasobem żądania i jego obecną funkcją.
- *CONNECT* – metoda używana do poinformowania serwera proxy o żądaniu stworzenia połączenia do wskazanego serwera,
- *OPTIONS* – metoda pozwalająca na wykrycie opcji komunikacyjnych dla żądanego zasobu.

- *TRACE* – metoda pozwalająca na żądanie by serwer odesłał z powrotem do klienta żądanie, które otrzymał. Żądanie nie może zawierać pola danych.

Według sekcji 3.2.1 RFC 7230, nazwy pól nagłówków (lub krótko nagłówków) powinny być zarejestrowane[5]. Do najbardziej popularnych nagłówków należą:

- *Accept* – określa typy MIME możliwe do akceptacji w odpowiedzi skierowanej do klienta,
- *Accept-Encoding* – wskazuje jakie typy kodowania danych klient wspiera,
- *Accept-Language* – wskazuje jakie języki naturalne klient preferuje,
- *Cache-Control* – zawiera instrukcje kontrolujące mechanizm cache'owania,
- *Connection* – kontroluje stan połączenia, np. czy ma pozostać otwarte,
- *Content-Encoding* – określa sposób kodowania danych przesyłanych w polu danych,
- *Content-Type* – wskazuje na typ MIME przesyłanego zasobu,
- *Content-Length* – określa rozmiar danych przesyłanych w polu danych,
- *Cookie* – zawiera ustawione wartości tzw. ciasteczek,
- *Host* – zawiera informacje o docelowej maszynie i porcie, nagłówek ten musi być obecny w żądaniach w wersji 1.1 protokołu,
- *Referer* – określa źródłowy adres URI, z którego pochodzi żądanie,
- *TE* – określa, jakie kodowania transferowe klient wspiera,
- *User-Agent* – wskazuje na aplikację wykonującą żądanie.

Zgodnie z RFC 7230 między nazwą nagłówka a dwukropkiem nie może być spacji ani tabulatora, jednak między dwukropkiem a wartością nagłówka, jak również między wartością nagłówka a końcem pola nagłówka, może być dowolna liczba takich znaków. Zazwyczaj przed wartością nagłówka jest tylko jedna spacja, a na końcu pola nie ma żadnych białych znaków. Kolejność nagłówków w żądaniu może być dowolna, ponieważ nie przypisano jej specjalnego znaczenia.

[5]Ich rejestr jest dostępny pod adresem: `https://www.iana.org/assignments/message-headers/message-headers.xhtml`

RFC 7230 definiuje również mechanizm tzw. pipeliningu. Przy zastosowaniu tego trybu komunikacji klient może wysyłać wiele żądań bez oczekiwania na powiązane z nimi odpowiedzi.

2.2 Złośliwe oprogramowanie

Istnieją różne podejścia klasyfikacji rodzajów złośliwego oprogramowania, różniące się kategoriami oraz sposobem ich definiowania (por. np. [4], [97]). Niemniej do najczęstszych typów można zaliczyć:

- konie trojańskie, podszywające się pod aplikacje niezłośliwe, by wykradać dane lub przejmować kontrolę nad maszynami,
- ransomware, szyfrujący dane użytkowników w celu wymuszenia okupu,
- botnety, tworzące zbiory maszyn wykorzystywanych do np. ataków,
- mobilne złośliwe oprogramowanie, infekujące systemy mobilne, np. telefony,
- spyware, zbierający informacje o użytkowniku bez jego wiedzy,
- adware, wyświetlający niechciane reklamy,
- rootkity, dające zdalny dostęp do maszyny w jak najmniej widoczny sposób,
- robaki internetowe, samokopiujące się np. przez sieć.

Patrząc z punktu widzenia sposobu kontroli nad złośliwym oprogramowaniem, ważnym zjawiskiem są botnety, czyli zbiory przejętych maszyn (botów) wykonujących polecenia otrzymane z serwera sterującego (Command and Control - C&C) [56]. Mimo że definicje botnetu bardziej skupiają się na możliwości wykorzystania przejętych maszyn do ataków lub działań na dużą skalę (por. [108]), to rozważania dotyczące aspektów ich komunikacji można wykorzystać także do innych typów złośliwego oprogramowania. Np. ransomware, mimo że nie zalicza się do kategorii botnetów, to może wykorzystywać komunikację protokołu HTTP do kontaktu z infrastrukturą przestępców w celu wymiany informacji [29]. Z punktu widzenia działań sieciowych mogą to być bardzo podobne operacje jak w przypadku botnetu, dlatego sposoby analizy i detekcji wykorzystujące ruch sieciowy używane przy analizie botnetów można wykorzystywać do analizy także i innych typów złośliwego oprogramowania.

Do łączności złośliwe oprogramowanie wykorzystuje najczęściej protokoły takie jak: IRC, HTTP/HTTPS, SMB, P2P oraz specjalnie stworzone protokoły bazujące na protokołach UDP lub TCP [28], [104]. W celu zmniejszenia szansy wykrycia, twórcy złośliwego oprogramowania wykorzystują także różne techniki ukrywania informacji, np. steganografię, sieciowe kanały ukryte oraz mimikrę [20]. Techniki te mogą wykorzystywać znane protokoły sieciowe w celu ukrycia obecności ruchu sieciowego złośliwego oprogramowania.

Analizę złośliwego oprogramowania można podzielić na trzy typy [27]:

- statyczną, w której próbka złośliwego oprogramowania nie jest wykonywana i dokonuje się analizy dostępnych cech pliku binarnego, np. entropii, a także inżynierii wstecznej (reverse engineering),
- dynamiczną, w której próbka złośliwego oprogramowania jest wykonywana i badane jest jej zachowanie, np. zmiany w rejestrze systemowym, wywołania API systemowego, generowany ruch sieciowy itp.,
- podejścia hybrydowe, łączące dwa powyższe typy.

Wykrywanie złośliwego oprogramowania bazuje na informacjach pochodzących z przeprowadzonych analiz. Wiele systemów wykorzystuje algorytmy uczenia maszynowego do wykrywania złośliwego oprogramowania [20], [101]. Systemy te bazują na różnych podejściach uczenia: nadzorowanego, nienadzorowanego oraz częściowo nadzorowanego. Używane są do tego cechy takie jak [101]:

- sekwencje bajtowe,
- wywołania systemowe oraz API,
- kody operacji,
- ruch sieciowy,
- informacje dotyczące systemu plików,
- rejestry procesora,
- cechy pliku wykonywalnego,
- łańcuchy znakowe.

Rozwój sieci neuronowych w latach 2010. umożliwił ich wykorzystanie także do detekcji złośliwego oprogramowania. Jak na razie trudno jednoznacznie określić czy sieci neuronowe są lepszym podejściem do detekcji złośliwego oprogramowania niż klasyczne metody uczenia maszynowego [20].

Innym sposobem opisu systemów wykrywania złośliwego oprogramowania jest określenie charakteru operacji detekcji. Na początku systemy wykorzystywały metody bazujące na sygnaturach, np. łańcuchach znakowych, które były poszukiwane w analizowanym pliku. Problemem takich metod jest konieczność aktualizacji sygnatur i zarządzanie nimi [20]. Systemy bazujące na analizie zachowania są kolejnym etapem rozwoju systemów analizy i detekcji złośliwego oprogramowania [20]. W podejściu tym monitoruje się zachowanie złośliwego oprogramowania i na tej bazie tworzy reguły wskazujące np. na wystąpienie anomalii. Z kolei systemy heurystyczne powstały na bazie systemów sygnaturowych i badających zachowanie. Wykorzystują techniki uczenia maszynowego w celu określenia odpowiednich cech i zachowania złośliwego oprogramowania, pomagających w jego wykryciu. Powstają także nowatorskie podejścia bazujące na monitorowaniu wykorzystania energii przez urządzenia oraz zainspirowane zjawiskami biologicznymi, np. algorytmami genetycznymi [20].

2.3 Użycie protokołu HTTP przez złośliwe oprogramowanie

Protokół HTTP używany jest przez złośliwe oprogramowanie do różnych celów:

- sprawdzenia łączności z internetem lub adresu IP,
- rejestracji w botnecie,
- pobrania komend z serwera C&C,
- pobrania dodatkowych modułów złośliwego oprogramowania,
- wysyłki danych zebranych o użytkowniku,
- pobrania dodatkowego złośliwego oprogramowania,
- wykonywania zleconych operacji, np. ataków DDoS lub generowania przychodu przez odwiedzanie linków reklamowych.

Obserwacja zachowania złośliwego oprogramowania wykonującego te cele umożliwia lepsze jego poznanie, co z kolei ułatwia jego detekcję oraz śledzenie. Proces obserwacji często

rozpoczyna się od ręcznej analizy ruchu sieciowego przez analityka złośliwego oprogramowania, dodatkowo ułatwionej jawną postacią protokołu HTTP [82], [103]. Analiza ta umożliwia stworzenie reguł do systemów wykrywania włamań sieciowych (NIDS), dzięki czemu można analizować ruch sieciowy na większą skalę. Niestety, w przypadku bazowania na regułach lub sygnaturach systemy te nie są wystarczające ze względu np. na konieczność częstej aktualizacji reguł [59].

Pomocą mogą być systemy bazujące na algorytmach uczenia maszynowego, które nie wymagają tworzenia reguł detekcji. Systemy te działają z wykorzystaniem różnych podejść. Wykrywanie złośliwego oprogramowania może odbywać przez użycie informacji o strukturze wiadomości [67], wartości nagłówków [57] lub przesyłanych danych [81]. Ich analiza może skupiać się na pojedynczych żądaniach [57], ich zestawach [74], a także wykorzystywać odpowiedzi serwera [85]. Pełny przegląd podejść do detekcji złośliwego oprogramowania wykorzystujących protokół HTTP znajduje się w rozdziale 3.

Jednym z ograniczeń poniższej rozprawy jest wykorzystanie do analizy tylko żądań protokołu HTTP. Odpowiedzi serwerów nie są brane pod uwagę, ponieważ nie zawsze są one dostępne do analizy, a także nie zawsze zawierają prawidłowe dane. Brak dostępności odpowiedzi może być spowodowany wyłączeniem serwera, z którym kontaktuje się analizowana próbka złośliwego oprogramowania, zmianą jego adresu, a także wykonywaniem analizy bez dostępu do internetu. W takim scenariuszu próbka jest analizowana w odizolowaniu, by nie mogła wykonać szkód innym użytkownikom internetu lub gdy konieczne jest zachowanie poufności i nieinformowanie właścicieli infrastruktury o dokonywanej analizie. Z kolei sytuacje, gdy odpowiedzi serwera nie zawierają prawidłowych danych spowodowane są przez:

- blokowanie ruchu wyjściowego przez urządzenia bezpieczeństwa i podmianę odpowiedzi, np. systemy Web Application Firewall w sieciach korporacyjnych [78],
- przejęcie infrastruktury przestępców przez badaczy bezpieczeństwa, np. przez fizyczne przejęcie lub sinkholing [56]. Wtedy otrzymana odpowiedź może się całkowicie różnić od tej wygenerowanej przez oryginalny serwer.
- wykorzystanie przez przestępców mechanizmu różnicowania odpowiedzi ze względu np. na źródłowy adres IP żądania. Może wtedy dochodzić do blokowania dostępu do prawdziwych zasobów przez podmianę odpowiedzi. Różnicowanie może być spowodowane ukierunkowaniem złośliwego oprogramowania na konkretny kraj lub chęcią utrudnienia

analizy badaczom bezpieczeństwa (por. [31]).

3 Przegląd aktualnego stanu wiedzy i jego związek z problemem postawionym w rozprawie

W poniższym rozdziale znajduje się przegląd aktualnego stanu wiedzy dotyczącego wykorzystania protokołu HTTP do identyfikacji i klasyfikacji złośliwego oprogramowania. Kolejne podrozdziały odpowiadają najważniejszym kwestiom rozpatrywanym w rozprawie, tzn.:

- charakterystyce ruchu sieciowego protokołu HTTP złośliwego oprogramowania,
- problemowi identyfikacji żądań HTTP złośliwego oprogramowania,
- wykrywaniu złośliwego oprogramowania przy użyciu protokołu HTTP,
- wykrywaniu złośliwego oprogramowania w scenariuszu otwartozbiorowym.

Każdy podrozdział zawiera omówienie najważniejszych prac, a także ich analizę krytyczną, obejmującą podsumowanie głównych zastrzeżeń oraz otwartych pytań.

3.1 Charakteryzowanie ruchu sieciowego protokołu HTTP złośliwego oprogramowania

Ruch sieciowy protokołów HTTP i DNS złośliwego oprogramowania był analizowany przez Rossowa i in. w systemie Sandnet [87]. Autorzy zbadali zachowanie ponad 100 tys. próbek złośliwego oprogramowania, z czego w 43,8% zaobserwowano ruch sieciowy. Analiza żądań HTTP wykazała, że 89,5% próbek użyło metody *GET*, natomiast 56,3% metody *POST*. Spośród 144 unikalnych nagłówków, nagłówek *User-Agent* wysłało 98,6% próbek, natomiast *Accept-Language* 44,3%.

Rozwinięciem analizy systemu Sandnet [87] jest analiza Nelsona [77]. Autor przedstawił w niej system do analizy złośliwego oprogramowania rozszerzony o wizualizację wyników. Został on wykorzystany do analizy zbioru danych pod kątem ruchu sieciowego z naciskiem na protokoły DNS i HTTP. Wśród badanych cech protokołu HTTP znalazły się metoda żądania, URI oraz wartości popularnych nagłówków. Analiza wykazała, że w 86,7% badanych plików występowała metoda *GET*, a w 86,8% metoda *POST*. Zaobserwowano 33 unikalne wartości nagłówka *User-Agent* oraz 24 różne nagłówki. Dodatkowo badania autora wykazały, że wartości nagłówków *Accept-Language* i *Content-Type*, mogą być użyte do identyfikacji ruchu sieciowego złośliwych aplikacji.

M. Calzarossa i in. w [19] dokonali analizy żądań HTTP wysyłanych do serwerów ich uniwersytetu. Zbadane zostały m.in. wersja protokołu, liczba nagłówków oraz wskazano, które były najpopularniejsze. Autorzy zauważyli, że liczba nagłówków oraz sposób ich użycia był różny między żądaniami przeglądarek i narzędzi automatycznych, co umożliwia łatwe ich odróżnienie.

Istnieją również źródła nieakademickie charakteryzujące żądania HTTP złośliwego oprogramowania. M. Mokbel w [75] prezentuje przykłady błędów w wiadomościach protokołu HTTP dokonywanych przez różne rodziny złośliwego oprogramowania, np. dodatkowe lub niedozwolone znaki czy niespójność wartości nagłówków z przesyłanymi danymi. M. Montoro przedstawił w [76] system oceny żądań HTTP. Dokonywana jest w nim analiza nagłówków, metody oraz URI dla określenia miary punktacyjnej złośliwości. Z kolei T. Lewis w [64] przedstawił zestaw anomalii mogących wskazywać na złośliwość komunikacji, np. niestandardowe wartości nagłówka *User-Agent*, długość URI lub błędy typograficzne w nagłówkach i ich wartościach. Wreszcie istnieją publiczne moduły analizy ruchu sieciowego w popularnym systemie sandboksowym Cuckoo[6] [23], [24]. Umożliwiają one analizę żądań HTTP pod kątem anomalii powiązanych z nagłówkiem *User-Agent* czy brakiem nagłówka *Referer* w żądaniach metody *POST*.

Do zaprezentowanych powyżej artykułów oraz źródeł nieakademickich można mieć kilka zastrzeżeń. Przede wszystkim liczba analizowanych cech jest ograniczona, a zależności między nimi nie są wystarczająco zbadane. Ograniczona jest również liczba zbadanych rodzin złośliwego oprogramowania, tym samym liczba żądań HTTP. Natomiast w przypadku źródeł nieakademickich z definicji twierdzenia w nich zawarte lub prezentowane cechy żądań nie zostały poparte rygorystyczną analizą akademicką. Zważywszy na ich potencjalną przydatność, warto podeprzeć je dowodem skuteczności. Analiza zawarta w rozdziale 4. stara się wypełnić te luki przez przeprowadzenie dogłębnej i systematycznej analizy różnych cech żądań HTTP na bazie dużej liczby rodzin złośliwego oprogramowania.

3.2 Identyfikacja żądań złośliwego oprogramowania

Według R. Bortolameotti i in. [15] protokół HTTP można wykorzystać do stworzenia modeli aplikacji używanych w monitorowanej sieci, które mogą posłużyć do odkrycia nieznanych aplikacji. Do tego celu służy ich system *DECANTeR*. Dla analizowanej maszyny tworzony jest odcisk (fingerprint) na bazie ruchu sieciowego protokołu HTTP, który tworzy pewien model jej zachowania. W sytuacji, gdy wykryte zostaną jakieś odstępstwa od tego modelu, system jest

[6] https://cuckoosandbox.org/

w stanie wskazać na obecność anomalii. Do stworzenia odcisku wykorzystuje się poklastrowane żądania *GET* i *POST*, z których wydobywa się informacje o wartości nagłówków *Host*, *User-Agent* i *Accept-Language*, nagłówkach obecnych we wszystkich żądaniach klastra, średnim rozmiarze żądania oraz ilości danych, które zostały wytransferowane przez żądania z klastra.

Z kolei R. Bortolameotti i in. w [16] zaprezentowali *HeadPrint* - system do detekcji anomalii w ruchu sieciowym bazujący na żądaniach protokołu HTTP. Żądania znanych aplikacji wykorzystywane są do stworzenia dwóch modeli: entropii par nagłówek-wartość oraz sekwencji nagłówków wysyłanych w żądaniach danej aplikacji. Działanie systemu przewiduje dwie fazy: treningu oraz właściwej detekcji. W pierwszej fazie system analizuje ruch sieciowy aplikacji w celu stworzenia jej odcisku, który jest używany w drugiej fazie jako baza porównawcza. W fazie detekcji żądania są analizowane i porównywane z odciskiem przy użyciu miary podobieństwa wartości modeli tworzących odcisk lub przy użyciu algorytmu uczenia maszynowego.

Z. Tang i in. w [100] zaproponowali metodę tworzenia odcisków z sesji HTTP różnych aplikacji. Na jej bazie został stworzony system *HSLF*. Zbiera on sekwencje par nagłówek-wartość żądań i odpowiedzi HTTP, które są następnie zamieniane na odcisk przy użyciu funkcji skrótu uwzględniającej lokalizację (local-sensitive hashing). Odciski dla znanych aplikacji są przechowywane w bazie danych. Przy analizie nowych sesji HTTP tworzony jest odcisk i porównywany z tymi przechowywanymi w bazie przy użyciu miary odległości Hamminga. Jeśli odcisk nie zostanie odnaleziony, zapisywany jest jako nieznany, by oczekiwać na identyfikację.

Polem badawczym związanym z identyfikacją złośliwego oprogramowania przy użyciu protokołu HTTP jest tworzenie odcisków palca przeglądarek internetowych. Pole to jest aktywnie badane (por. P. Laperdrix i in. [63]), także w kierunku prywatności użytkowników internetu. Odciski przeglądarek internetowych tworzone są na bazie różnych cech przy użyciu technik aktywnych i pasywnych. Techniki aktywne badają między innymi środowisko działania przeglądarki, system operacyjny, rozdzielczość ekranu, listę zainstalowanych wtyczek i czcionek, itd. Z kolei w podejściu pasywnym badane są żądania wysyłane przez przeglądarkę, np. wartości popularnych nagłówków czy ich kolejność.

Do identyfikacji ruchu sieciowego mogą zostać wykorzystane także inne protokoły. Szyfrowany odpowiednik protokołu HTTP, czyli HTTPS, może być wykorzystany do rozpoznania aplikacji go wykorzystującej (por. W. Shbair i in. [93]). Na bazie protokołu SMTP można stworzyć dialekty klientów pocztowych, dzięki którym można zidentyfikować rodzinę złośliwego oprogramowania wysyłającą niechciane wiadomości e-mail (P. Bazydło i in. [6], G. Stringhini

i in. [98]). Z kolei protokół DNS może posłużyć do wykrycia botów w botnecie (A. Blaise i in. [14]), detekcji ataków DDoS (C. Fachkha in. [34]) lub identyfikacji serwerów DNS (T. Kim i in. [60]). Z kolei O. Segal i in. w raporcie technicznym [92] omówili sposoby identyfikacji aplikacji wykorzystujących protokół HTTP/2.

Tworzenie odcisków może być również dokonywane w oderwaniu od konkretnych protokołów aplikacyjnych w sposób bardziej ogólny. T. van Ede i in. w [30] zaprezentowali system do tworzenia odcisków aplikacji mobilnych przy użyciu ich zaszyfrowanego ruchu sieciowego. Ruch ten jest klastrowany przy użyciu zdefiniowanego zestawu cech bazujących na zależnościach czasowych, wykorzystywanych adresach i rozmiarach danych. Z otrzymanych klastrów izolowane są przeglądarki internetowe i przetwarzane w sposób rozszerzony. Następnie klastry są korelowane krzyżowo pod względem czasu oraz adresu docelowego w celu stworzenia grafów korelacji. Na koniec tworzony jest odcisk będący zestawem adresów docelowych będących tworzących maksymalne kliki w otrzymanych grafach. W celu określenia czy odciski są do siebie podobne, a tym samym rozpoznać aplikację, są one porównywane na bazie współczynnika podobieństwa Jaccarda.

Z kolei J. Holland i in. zaproponowali w [52] metodę na reprezentację ruchu sieciowego w ustandaryzowany sposób. Odcisk tworzony jest dla każdego pakietu przez mapowanie bajtów jego wartości na wektor cech reprezentujący wszystkie możliwe nagłówki podstawowych protokołów internetowych takich jak IP, TCP, UDP i ICMP. Co ważne, mapowanie danych przenoszonych przez protokoły wyższych warstw niż te wymienione musi być zdefiniowane przez użytkownika. Według autorów reprezentacja tworzona przez ich narzędzie w łatwy sposób może zostać użyta przez algorytmy uczenia maszynowego, umożliwiając tym samym identyfikację aplikacji, systemu operacyjnego lub urządzeń.

W [66] J. Li i in. zaprezentowali system do identyfikacji aplikacji mobilnych rozszerzony o możliwość śledzenia działań użytkownika. Dodatkowo do rozpoznawania aplikacji autorzy zastosowali podejście otwartego świata (open-world). W czasie fazy identyfikacji autorzy dopuszczają możliwość wystąpienia ruchu aplikacji, która była nieznana w czasie treningu. Do tworzenia odcisków wykorzystywane są przepływy zaszyfrowanego ruchu aplikacji. Definicja otwartego świata podana przez autorów jest bardzo podobna do podejścia klasyfikacji otwartozbiorowej, niemniej autorzy dokonują rozpoznawania nieznanych aplikacji tylko po to, by obniżyć współczynnik klasyfikacji fałszywie pozytywnych, nie wykorzystując faktu identyfikacji klas nieznanych do rozszerzenia listy rozpoznawanych aplikacji.

Tabela 3.1 podsumowuje artykuły omówione w powyższej części ze względu na scenariusz użycia.

Cel użycia	Artykuł
Identyfikacja aplikacji klienckiej lub użytkownika	P. Laperdrix i in. [63], O. Segal i in. [92], J. Li i in. [66], T. van Ede i in. [30], Z. Tang i in. w [100]
Wykrywanie nieznanych aplikacji	R. Bortolameotti i in. [15], [16]
Identyfikacja usługi lub serwera	W. Shbair i in. [93], T. Kim i in. [60]
Identyfikacja rodziny złośliwego oprogramowania	G. Stringhini i in. [98], P. Bazydło i in. [6]
Wykrywanie złośliwego oprogramowania	A. Blaise i in. [14]
Detekcja ataków	C. Fachkha i in. [34]
Uogólniona identyfikacja ruchu sieciowego	J. Holland i in. [52]

Tabela 3.1: Klasyfikacja artykułów dotyczących wykorzystania ruchu sieciowego do identyfikacji pod względem scenariusza użycia.

Tylko część z zaprezentowanych powyżej podejść wykorzystuje protokół HTTP, na którym skupia się niniejsza rozprawa. Pośród nich, systemy DECANTeR, HeadPrint oraz HSLF zostały zaprojektowane do tworzenia pewnej bazowej reprezentacji zachowania aplikacji w sieci, w której zostały uruchomione. W tym celu wykorzystywane są zestawy żądań lub zestawy żądań i odpowiedzi. Minusem takiego podejścia jest konieczność wykonania fazy tworzenia modelu monitorowanej sieci, bez czego systemy te nie będą w pełni działały. Dodatkowo systemy te nie analizują danych przesyłanych np. w adresie URL lub w polu danych. Z kolei w podejściach tworzących odcisk przeglądarek internetowych celem jest dokładna identyfikacja tylko tego jednego rodzaju aplikacji. Wykorzystywane do tego techniki aktywne nie mogą być zastosowane do analizy złośliwego oprogramowania ze względu na częste ograniczenia środowisk badawczych. Przykładowo w systemach sandboksowych unika się ingerencji w zachowanie analizowanych próbek złośliwego oprogramowania. System nPrint umożliwia tworzenie ustandaryzowanych odcisków ruchu sieciowego, niemniej jego autorzy skupili się na protokołach nieaplikacyjnych. Możliwe jest tworzenie odcisku dla protokołu HTTP, ale definicja jego tworzenia musi być stworzona samodzielnie. Wreszcie analiza przedstawiona przez O. Segala i in. [92] skupia się tylko na protokole HTTP w wersji 2, przez co nie da się jej bezpośrednio porównywać do podejść bazujących na starszych wersjach tego protokołu.

Powyższe problemy stara się rozwiązać system identyfikacji żądań HTTP o nazwie *Hfinger*, zaprezentowany w rozdziale 5, który został opracowany przez autora niniejszej rozprawy. Dokonuje on identyfikacji pojedynczych żądań HTTP złośliwego oprogramowania przy użyciu szerokiego zakresu danych dostępnych w różnych częściach wiadomości: adresie URL, nagłów-

kach oraz przesyłanych danych. Ponadto, wykorzystuje do tego celu pasywną analizę na bazie plików pcap i nie wymaga tworzenia modeli zachowania aplikacji w monitorowanej sieci.

Oprócz podejść akademickich istnieją także narzędzia umożliwiające identyfikację żądań HTTP przez tworzenie ich odcisku. Są to FATT[7], Mercury[8] oraz p0f[9]. Narzędzia te dokonują pasywnej analizy bazującej na plikach pcap. Inne rozwiązania wykorzystują także protokół HTTP, ale skupiają się na innych zadaniach. Przykładowo FingerprintJS[10], który wykorzystuje metody aktywne do identyfikacji tylko przeglądarek internetowych, a także różne systemy do identyfikacji serwerów, np. httprecon[11], httprint[12] lub nmap[13].

3.3 Wykrywanie złośliwego oprogramowania przy użyciu protokołu HTTP

W poniższym podrozdziale zamieszczono przegląd najważniejszych prac dotyczących wykrywania złośliwego oprogramowania przy użyciu protokołu HTTP. Mimo że powiązane tematycznie przez wykorzystanie tego samego protokołu, w rozdziale tym nie będą omawiane kwestie wykrywania tuneli HTTP oraz wykrywania przy użyciu systemów IDS ataków na aplikacje przeglądarkowe. Tunele HTTP używane są do przesyłu danych w różnych celach, w tym złośliwych, ale nie są one bezpośrednio połączone ze złośliwym oprogramowaniem. Z kolei ataki na aplikacje przeglądarkowe, wykrywane przez systemy IDS (czasami zwane także Web Application Firewall), obejmują całą gamę operacji mających na celu przełamanie zabezpieczeń i pozyskanie danych z serwisów komunikujących się przez protokół HTTP. Pośród tych ataków mogą znajdować się działania złośliwego oprogramowania, jednak domena detekcji jest znacznie szersza i obejmuje także operacje wykonywane przez wyspecjalizowane automatyczne narzędzia oraz ręcznie tworzone ataki. Systemy te wykorzystują głównie cechy URL, wartości nagłówków oraz danych przesyłanych w żądaniach (por. N. Agarwal i in. [2]), więc są to cechy podobne do tych używanych przez systemy detekcji malware'u. Niemniej jednak niniejsza rozprawa skupia się tylko na złośliwym oprogramowaniu w związku z tym nie zamieszczono tu przeglądu prac o systemach IDS. Jest to bardzo szeroki i w dużej części odrębny obszar badawczy. Zainteresowane osoby mogą przeczytać więcej o tych systemach np. w H. Hindy i in.

[7] https://github.com/0x4D31/fatt/
[8] https://github.com/cisco/mercury
[9] https://lcamtuf.coredump.cx/p0f3
[10] https://github.com/fingerprintjs/fingerprintjs
[11] https://www.computec.ch/projekte/httprecon/
[12] https://net-square.com/httprint.html
[13] https://nmap.org/

[51].

Poniższy przegląd aktualnego stanu wiedzy został przeprowadzony ze względu na: cel analizy, elementy ruchu sieciowego wykorzystywane w niej, podejście do procesu detekcji, deklarowaną możliwość detekcji niewidzianych wcześniej rodzin oraz wykorzystanie żądań/odpowiedzi HTTP.

Cele analizy w wykrywaniu złośliwego oprogramowania są różne. Może nimi być grupowanie podobnych zachowań, przykładowo poprzez klastrowanie, w celu stworzenia reguł do systemów IDS. Może to być klasyczna detekcja, tzn. określenie czy jakieś zachowanie jest złośliwe. Wreszcie może być to klasyfikacja określająca przynależność do jednej z wielu rodzin złośliwego oprogramowania.

Klastrowanie używane jest do grupowania podobnych żądań malware'u. Celem takiego grupowania może być tworzenie reguł na potrzeby systemów IDS. Właśnie w tym kierunku przeprowadzili badania R. Perdisci i in. w [80]. Stworzyli system klastrujący żądania pod względem m.in. cech adresu URL i wykorzystanych metod żądań. Na bazie utworzonych klastrów automatycznie generowane są reguły IDS. System stworzony przez N. Kheira i in. [58] także wykorzystuje klastrowanie do tworzenia reguł detekcji. Autorzy skupili się na wykorzystaniu adresów URL, na których bazie dokonują kilku etapowego grupowania żądań. Jego wynikiem jest stworzenie sygnatury URL, której można użyć do detekcji złośliwego oprogramowania. Z kolei A. Zarras i in. w [115] wykorzystali klastrowanie do tworzenia szablonów żądań ze względu na wartość nagłówków. W sytuacji, gdy analizowane żądanie jest podobne do złośliwego szablonu, może także zostać uznane za złośliwe.

W dalszej części przeglądu znajdują się przykłady systemów dokonujących detekcji lub klasyfikacji.

Podstawą do analizy przy detekcji złośliwego oprogramowania mogą być różne elementy ruchu sieciowego protokołu HTTP. Najczęściej wykorzystywany jest URL, metoda żądania/odpowiedź serwera, nagłówki i ich wartości, a także dane przesyłane w żądaniu lub odpowiedzi. Podczas gdy istnieją metody wykorzystujące do detekcji pojedyncze elementy np. tylko URL (B. Q i in. [83]) lub tylko wartość pojedynczego nagłówka, np. *User-Agent* (N. Kheir [57]), wiele podejść wykorzystuje kilka takich elementów (np. K. Li i in. [67], X. Wu i in. [109], R. Perdisci i in. [80]).

Do detekcji złośliwego oprogramowania wykorzystywana jest również wartość nagłówka *User-Agent*. N. Kheir w [57] analizuje nagłówek *User-Agent* by przy pomocy jego wartości

dokonać dwuetapowej klasteryzacji żądań. Wykorzystywanymi cechami są długość łańcucha znakowego, częstość występowania znaków, ale także podobieństwo fragmentów. Otrzymane grupy żądań wykorzystywane są do wygenerowania sygnatur umożliwiających detekcję złośliwego oprogramowania. M. Grill i in. w [44] zaprezentowali system wykrywający rozbieżności w wartościach nagłówka *User-Agent*. System ten analizuje ruch badanej sieci i tworzy model występujących wartości opisywanego nagłówka dla poszczególnych maszyn. W sytuacji gdy napotka nieznaną rzadko występującą wartość, brak tego nagłówka dla rzadko odwiedzanej domeny lub stwierdzi rozbieżność w wartości już poznanej, alarmuje o wystąpieniu anomalii.

Wykrywanie złośliwego oprogramowania może odbywać się poprzez wykorzystanie cech protokołów HTTP i DNS. W systemie detekcji A. Udiyono i in. w [102] wykorzystali cechy bazujące na żądaniach HTTP, m.in. rozmiar pola danych czy wartość nagłówka *User-Agent*, ale także bazujące na DNS jak np. liczba zapytań DNS czy liczba odpowiedzi *NXDomain*. Z kolei M. N. Sakib i in. w [89] wykorzystują dwuetapową analizę ruchu sieciowego w celu wykrycia komunikacji z serwerem C&C. W pierwszej fazie URL żądań HTTP analizowany jest przez nienadzorowany algorytm detekcji anomalii. Natomiast w fazie drugiej analizowane są odpowiedzi DNS przez półnadzorowany algorytm detekcji anomalii.

Część z prac do wykrywania złośliwego oprogramowania wykorzystuje informacje o połączeniach w warstwach niższych niż protokół HTTP. Przykład takiego podejścia został zaproponowany przez F. Haddadi i in. w [46]. Autorzy do detekcji wykorzystują cechy bazujące na przepływach, które są danymi wejściowymi dla klasycznego algorytmu uczenia maszynowego. Istnieją przykłady systemów wykrywających zagrożenia także dla innych protokołów niż HTTP, ale również korzystające z właściwości przepływów, jak np. F. Haddadi i in. w [47]. Cechy bazujące na przepływach mogą być wykorzystywane obok tych korzystających z żądań i odpowiedzi HTTP, np. wersji protokołu, typu żądania lub kodu odpowiedzi, jak u P. A. A. Resende w [85].

Innym podejściem do wykrywania złośliwego oprogramowania jest wykorzystanie analizy czasowej protokołu HTTP. M. Eslahi i in. w [33] zaprezentowali rozwiązanie bazujące na analizie wzorów okresowości. Wykorzystując żądania *GET* i *POST*, proponowany system analizuje zachowanie maszyn w zdefiniowanych przedziałach czasowych. Celem jest odkrycie wzorów świadczących o okresowym wysyłaniu żądań, a także ich częstości i interwałach między nimi. Z kolei w [32] M. V. Naseri i in. zaproponowali analizę histogramową przebiegów czasowych sesji używających metod *GET* i *POST*. Rozwiązanie to modeluje rozmiar przesyłanych danych w czasie, czas między wysyłką żądań, a także porównuje zachowanie badanej maszyny w różnych

okresach. Autorzy zaproponowali wykorzystanietcyh cech w klasycznych algorytmach uczenia maszynowego takich jak C4.5 czy lasy losowe.

W celu wykrycia złośliwego oprogramowania stosowane są różne podejścia do procesu detekcji: (i) heurystyczne, wykorzystujące pewne zdefiniowane reguły, (ii) wykorzystujące algorytmy uczenia maszynowego do tworzenia modeli danych, oraz (iii) wykorzystujące sieci neuronowe do przetwarzania danych i klasyfikacji.

Podejście heurystyczne wykorzystuje pewne reguły lub zaobserwowane anomalie do wykrywania złośliwego oprogramowania. Przykładowo Z. Li i in. w [68] detekcję opierają na niespójności typu MIME zadeklarowanego w nagłówku *Content-Type* żądań oraz w przenoszonych danych. Dosyć podobną zasadę stosują F. Xu i in. w [111], którzy proponują wykorzystanie niespójności rozszerzenia pliku w adresie URL żądania lub wartości nagłówka *Content-Type* w odpowiedzi z rzeczywistym typem danych przesyłanych przez serwer. Według badań autorów jeśli w rzeczywistości przesyłane dane są w formacie wykonywalnym PE (Portable Executable) i zadeklarowany typ danych jest obrazem lub popularnym plikiem biurowym, to w ponad 99% przypadków przesyłany plik jest złośliwym oprogramowaniem.

Innym podejściem do procesu detekcji jest wykorzystanie algorytmu uczenia maszynowego do stworzenia modelu ruchu protokołu HTTP umożliwiającego odróżnienie złośliwego oprogramowania od tego bezpiecznego. Na wejściu algorytmów stosowane są specjalnie zaprojektowane cechy bazujące na elementach ruchu sieciowego analizowanego protokołu. Przykładowo S. Mizuno i in. w [74] zaproponowali system do detekcji złośliwego oprogramowania wykorzystujący informacje z URL oraz wartości nagłówków do klastrowania żądań HTTP. Pogrupowanie na bazie miary prawdopodobieństwa warunkowego obecności danej części nagłówka w konkretnym jego miejscu umożliwia stworzenie szablonów żądań. Są one wykorzystywane jako dane wejściowe dla właściwego detektora bazującego na wybranym algorytmie uczenia maszynowego (lasach decyzyjnych lub maszynie wektorów nośnych) lub prostej sieci neuronowej. Z kolei K. Li i in. w [67] wykorzystują do detekcji cechy adresu URL, wartości nagłówków *Content-Type* i *User-Agent*, kod odpowiedzi serwera oraz sekwencję nagłówków. Do klasyfikacji użyty został algorytm lasu losowego oraz XGBoost.

Wraz z rozwojem badań nad sieciami neuronowymi, zaczęto ich używać do wykrywania złośliwego oprogramowania, w tym także przy użyciu protokołu HTTP. Sieci neuronowe umożliwiają bezpośrednie użycie przesyłanych danych bez konieczności projektowania i przetwarzania cech jak ma to miejsce w klasycznych algorytmach uczenia maszynowego. W [70] Z. Liu i

in. zamieniają na obraz w skali szarości dane pakietów składające się na sesję HTTP. Dzięki temu staje się możliwe przetwarzanie ruchu sieciowego jako obrazu przy użyciu generatywnej sieci przeciwstawnej (Generative Adversarial Networks — GAN). Z kolei J. Xie i in. w [110] wykorzystują projektowane cechy jak i surowe dane. Ich system analizuje osobno żądania i odpowiedzi HTTP w celu uzyskania cech statystycznych pakietów, cech przepływów, a także bezpośrednio przesyłanych danych. Informacje te zasilają sieć neuronową o hybrydowej architekturze, w której surowe dane są transformowane przy użyciu podsieci splotowej, by ostatecznie z resztą zakodowanych danych trafić do sieci bazującej na komórkach długiej pamięci krótkotrwałej (Long Short-Term Memory — LSTM). Istnieją także rozwiązania nieskupiające się jedynie na protokole HTTP, lecz wykorzystujące informacje o przepływach oraz dane przesyłane w warstwie aplikacji bez względu na użyty protokół. Używane są do tego architektury bazujące na sieciach splotowych (Convolutional Neural Networks - CNN), przykładowo, P. Xu i in. [112], G. Marín i in. [72], W. Wang i in. [107], S. Taheri i in. [99]. Wykorzystywana może być przy tym transformacja danych do postaci obrazu w skali szarości.

Powyższe prace skupiają się na standardowych platformach lub nie określają typu analizowanych sieci. Niemniej jednak część z tych rozwiązań skupia się na wykrywaniu złośliwego oprogramowania na platformach mobilnych. Z. Li i in. w [69] zaprezentowali system do wykrywania złośliwego oprogramowania. Do analizy wykorzystywane są wartości popularnych nagłówków oraz adresy URL żądań, na bazie czego obliczany jest współczynnik złośliwości. Bierze on pod uwagę częstość występowania danej wartości i jej obecność w bazie znanych wartości. Współczynnik określa czy żądanie jest złośliwe i do jakiej rodziny należy. Dla zwiększenia wydajności klasyfikacji żądania są dodatkowo klastrowane przy użyciu adresu URL w celu lepszej estymacji współczynnika złośliwości.

X. Wu i in. w [109] opisali system wykrywania przepakowywania aplikacji na system operacyjny Android. Autorzy rozumieją to jako dodanie dodatkowego kodu do aplikacji publikowanych jako bezpieczne i niezłośliwe. Działanie systemu polega na stworzeniu modelu żądań HTTP wysyłanych przez różne aplikacje, który bazujące na metodzie żądania, adresie URL oraz obecności nagłówków przekierowujących żądania. Po stworzeniu modelu system może analizować żądania w celu określenia czy część z nich jest podobna do tych generowanych przez znane, niezłośliwe aplikacje, oraz czy występują jakieś dodatkowe żądania świadczące od przepakowaniu.

H. Bai i in. w [3] wykorzystują reprezentację tekstową danych warstwy aplikacji, w tym

protokołu HTTP, do wykrywania złośliwego oprogramowania na platformach mobilnych przy użyciu złożonej sieci neuronowej. Dane przenoszone w przepływach są zamieniane na postać tekstową, a następnie transformowane do postaci n-gramów przy użyciu odpowiedniej architektury sieci CNN. W kolejnym etapie przepływy reprezentowane są przy użyciu zestawu GRU, by ostatecznie zostać przekazane do warstw wyjściowych umożliwiających klasyfikację.

Problem wykrywania nowego, wcześniej nieznanego złośliwego oprogramowania przy użyciu protokołu HTTP był rozwiązywany na kilka sposobów. D. Bekerman i in. w [7] zaprezentowali system detekcji bazujący na analizie 972 cech ruchu sieciowego różnych protokołów, w tym HTTP. Po przeprowadzeniu wyboru cech, 12 z nich, obejmujące także medianę czasu między wysyłką żądań HTTP, użyto do zbudowania modeli bazujących na trzech algorytmach uczenia maszynowego: naiwnym klasyfikatorze bayesowkim, J48 i lesie losowym. Badanie wykrywania nieznanych rodzin dokonano przez wykluczenie jednej rodziny ze zbioru danych do celów treningu, by następnie dodać ją w fazie testów. Jako że oceny eksperymentalnej dokonywano w schemacie klasyfikacji binarnej, nie odpowiada to scenariuszowi otwartozbiorowemu, w którym dokonywana jest klasyfikacja do wielu rodzin.

Z kolei K. Bartos i in. w [5] zaprezentowali system do detekcji nieznanego złośliwego oprogramowania, a także nowych wariantów już znanych rodzin. Rozwiązanie to wykorzystuje cechy bazujące na przepływach oraz adres URL żądań, które pogrupowane, są następnie transformowane do postaci odpowiednich histogramów, tworzonych przy użyciu samopodobieństwa (self-similarity) grup. Zamieniane są one następnie na ostateczny wektor cech, które są nieczułe na zmiany w ruchu sieciowym. Tak stworzony wektor cech umożliwia dokonanie klasyfikacji binarnej przy użyciu algorytmu SVM i, według autorów, wykrycie także nowych wariantów lub nieznanego złośliwego oprogramowania. Jest to kolejne podejście niespełniające warunku scenariusza otwartozbiorowego, tzn. klasyfikacji do wielu rodzin.

Systemy wykorzystujące klastrowanie także potencjalnie mogą wykrywać nieznane rodziny. Ich przykłady zostały zaprezentowane już powyżej: R. Perdisci i in. [80], N. Kheir i in. [58], A. Zarras i in. [115], Z. Li i in. [69]. Systemy te umożliwiają powiązanie ze sobą ruchu sieciowego należącego do jednej rodziny złośliwego oprogramowania, jednocześnie odseparowując go od ruchu innych rodzin praktycznie bez wcześniejszej jego znajomości. Niemniej jednak, jako że klastrowanie jest metodą nienadzorowaną, to z jednej strony konieczne jest oznaczenie klastrów, tzn. przypisanie informacji o przynależności do konkretnej rodziny złośliwego oprogramowania. Z drugiej strony nowo odkryte grupy, które są oddzielne od tych już oznaczonych, będą musiały

przejść przez proces decyzyjny czy jest to nowy wariant już znanej rodziny, czy też całkowicie nowa rodzina.

Podejścia tworzące modele standardowych, niezłośliwych zachowań (np. M. Grill i in. [44], M. N. Sakib i in. w [89]) jako złośliwe lub podejrzane oznaczają wszystkie zachowania odstające od tych modeli. W ten sposób mogą wykrywać nowe zagrożenia, ale tylko jeśli będą nadzorowane przez operatora, inaczej nie są w stanie oznaczyć, że wykryta rodzina złośliwego oprogramowania jest nieznana.

W procesie analizy ruchu sieciowego protokołu HTTP mogą być wykorzystywane same żądania, jak i żądania i odpowiedzi. Określenie, jaki rodzaj wiadomości wykorzystywany jest w opisywanych rozwiązaniach zawarto w tabeli 3.2, gdzie znajduje się podsumowanie powyższego przeglądu.

Artykuł	Cel analizy	Przedmiot analizy w protokole HTTP	Mechanizm detekcji	Detekcja nieznanych	Wykorzystanie żądań / odpowiedzi HTTP
[3]	klasyfikacja	surowe dane warstwy aplikacji	CNN + GRU	nie	żądania i odpowiedzi
[5]	detekcja	przepływ + URL	SVM	tak	żądania i odpowiedzi
[7]	klasyfikacja	wiele protokołów	NB, RF, J48	tak	wiele protokołów
[32]	detekcja	zależności czasowe	C4.5, RF, NB, SVM, FNN	nie	żądania i odpowiedzi
[33]	detekcja	zależności czasowe	reguły heurystyczne	nie	żądania
[44]	detekcja	wartość User-Agent	reguły heurystyczne	częściowo	żądania
[46]	detekcja	przepływ	C4.5, NB	nie	przepływ
[47]	detekcja	przepływ	C4.5, sieci bayesowskie, algorytm genetyczny	nie	przepływ
[57]	gen. reguł detekcji	wartość User-Agent	klastrowanie k-średnich	częściowo	żądania
[58]	gen. reguł detekcji	URL	k-średnie + DBScan	częściowo	żądania
[67]	detekcja	URL + wartości i kolejność nagłówków	RF, XGBoost	nie	żądania i odpowiedzi
[68]	detekcja	wartość nagłówka + przesyłane dane	reguły heurystyczne	częściowo	żądania
[69]	klasyfikacja	URL + wartości nagłówków	klastrowanie SL	częściowo	żądania
[70]	klasyfikacja	surowe dane sesji	GAN	nie	żądania i odpowiedzi
[72]	klasyfikacja	surowe dane pakietów lub przepływów	CNN + LSTM	nie	przepływ
[74]	detekcja	URL + wartość nagłówków	SVM + RF + DNN	nie	żądania
[80]	gen. reguł detekcji	URL + metoda + przesyłane dane	BIRCH + klastrowanie SL	nie	żądania i odpowiedzi
[83]	klasyfikacja	URL	RF + liniowy SVM + LR + DT + SGD	nie	żądania
[85]	detekcja	przepływy + wartości nagłówków + metoda + kod odpowiedzi + zależności czasowe	RF	nie	przepływy + żądania i odpowiedzi

[89]	detekcja	URL	Nierówność Czebyszewa + OC-SVM + NN & LOF	tak	żądania HTTP + odpowiedzi DNS
[99]	detekcja	surowe dane warstwy aplikacji	CNN	nie	protokoły warstwy aplikacji
[102]	detekcja	wartość nagłówków + przesyłane dane	DT + RF + kNN	nie	żądania HTTP + DNS
[107]	klasyfikacja	surowe dane wszystkich warstw lub warstwy aplikacji	CNN	nie	protokoły warstwy aplikacji
[109]	detekcja	URL + metoda + nagłówki odpowiedzi	grupowanie VPT	nie	żądania i odpowiedzi
[110]	detekcja	URL + metody + wartości nagłówków + kody odpowiedzi + przesyłane dane + surowe dane	CNN + LSTM	nie	żądania i odpowiedzi
[111]	detekcja	URL + wartość nagłówków + dane z odpowiedzi	reguły heurystyczne	tak	żądania i odpowiedzi
[112]	klasyfikacja	surowe dane przepływu	CNN + LSTM	nie	przepływ
[115]	klasyfikacja	URL + kolejność i wartości nagłówków	klastrowanie	tak	żądania

Tabela 3.2: Podsumowanie omówionych artykułów dotyczących wykrywania złośliwego oprogramowania przy użyciu protokołu HTTP. Użyte skróty: RF (Random Forest), SVM (Support Vector Machines), LR (Linear Regression), DT (Decisision Tree), SGD (Stochastic Gradient Descent), NB (Naive Bayes), GAN (Generative Adversarial Network), CNN (Convolutional Neural Network), LSTM (Long Short-Term Memory), GRU (Gated Recurrent Unit), OC-SVM (One Class Support Vector Machines), VPT (Vantage-Point Tree), kNN (k Nearest Neighbors), FNN (Forward Neural Network), DNN (Deep Neural Network), NN (Nearest Neighbor), LOF (Local Outlier Factor), SL (Single-Linkage).

Zaprezentowany powyżej przegląd aktualnej wiedzy o wykrywaniu złośliwego oprogramowania przy użyciu protokołu HTTP wskazuje na kilka problemów. Nie wszystkie zaproponowane rozwiązania umożliwiają przypisanie do wielu klas złośliwego oprogramowania - skupiają się na klasyfikacji binarnej (np. [32], [44], [99]). Podczas gdy stwierdzenie złośliwości ruchu sieciowego jest wartościową informacją, to mimo wszystko nie umożliwia pełnego osądu sytuacji, który jest dostępny w przypadku określenia, jaka rodzina została wykryta. Podobnie jest z wykorzystaniem informacji dostępnych w przesyłanych danych - tylko niektóre metody wykorzystują informacje z wszystkich elementów wiadomości HTTP: wartości nagłówków, ich miejsca w wiadomości, informacji sterujących (np. metody żądania lub kodu odpowiedzi) czy przesyłanych danych (np. [67], [110], [70]). Rezygnacja z tych informacji potencjalnie zmniejsza możliwości dyskryminacyjne tworzonych klasyfikatorów. Z kolei takie podejścia jak np. [3], [32] czy [80] wymagają do działania zarówno żądań jak i odpowiedzi HTTP. Odpowiedzi

serwera nie zawsze są dostępne do analizy lub mogą nie być wiarygodne, np. ze względu na wyłączenie infrastruktury przestępców lub jej przejęcie przez badaczy bezpieczeństwa. Z tego powodu wykorzystanie odpowiedzi serwera może prowadzić do tworzenia błędnych modeli detekcji. Wreszcie, niektóre systemy według deklaracji autorów umożliwiają wykrywanie ruchu sieciowego nieznanych rodzin złośliwego oprogramowania. Te rozwiązania, które bazują na pewnej cesze dyskryminatywnej lub wykrywaniu anomalii (np. [44], [89]) rzeczywiście mogą dokonać poprawnej detekcji, ale nie są na tyle wyspecjalizowane, by poinformować, że jest to całkowicie nowa rodzina. Także systemy dokonujące detekcji wieloklasowej (np. [72], [7]) nie są wyposażone w mechanizmy stwierdzające, że analizowana klasa jest nieznana. Potencjalnie najbliżej rozwiązania omawianego problemu są systemy wykorzystujące klastrowanie, jak np. [80], [58], [115], [69]. Dzięki grupowaniu systemy te mogą odkryć nieznane rodziny złośliwego oprogramowania. Niestety wadami tych systemów jest konieczność rozpoznawania rodzin w stworzonych grupach w celu późniejszej detekcji, a w sytuacji stworzenia nowego klastra odrębnego od reszty określenie czy jest to całkowicie nowa rodzina, czy jednak nowy klaster rodziny już znanej.

Niestety żadne z omawianych rozwiązań nie spełnia wymagań scenariusza otwartozbiorowego, tzn. nie dokonuje detekcji wieloklasowej oraz nie wykrywa klas nieznanych. Z tego powodu oraz ze względu na problemy przedstawione powyżej, zdecydowano się na stworzenie systemu detekcji wieloklasowej, który wykorzystywałby różne elementy żądań HTTP, umożliwiając przy tym działanie zgodnie ze scenariuszem otwartozbiorowym. System ten został zaprezentowany w rozdziale 6. Jako że według wiedzy autora jest to pierwszy system otwartozbiorowy nastawiony na wykrywanie złośliwego oprogramowania przy użyciu tylko protokołu HTTP, w pierwszym kroku dokonano ogólnego przeglądu systemów szeroko pojętego bezpieczeństwa informacji działających według scenariusza otwartozbiorowego. Przegląd ten znajduje się w rozdziale 3.4.

3.4 Wykrywanie złośliwego oprogramowania w scenariuszu otwartozbiorowym

W poniższym podrozdziale zaprezentowano różne sposoby zastosowania scenariusza klasyfikacji otwartozbiorowej do problemów detekcji nieznanych ataków sieciowych przez systemy wykrywania włamań lub do detekcji złośliwego oprogramowania. Dla porównania opisano również podejścia obejmujące problem identyfikacji nieznanego ruchu sieciowego, który jest blisko powiązany z problemem identyfikacji złośliwego oprogramowania na bazie ruchu sieciowego.

Ostatnie lata przyniosły wiele artykułów prezentujących badania problemu detekcji nieznanych ataków sieciowych przez systemy wykrywania włamań. Wiele z nich wykorzystuje do analizy wyspecjalizowaną warstwę sieci neuronowej OpenMax, zaprezentowaną przez A. Bendale i in. w [9].

J. Henrydoss i in. w [50] zaproponowali wykorzystanie maszyny wektorów ekstremalnych (EVM) do wykrywania włamań sieciowych. Do celów porównawczych wykorzystano klasyfikator bazujący na maszynach wektorów nośnych kalibrowanych rozkładem Weibulla (W-SVM). Oba algorytmy zostały przebadane na danych pochodzących ze zbioru danych KDDCUP'99 dając podobne wyniki pod względem dokładności dla różnych wartości otwartości zbioru danych.

Następnie, S. Cruz i in. w [22] zaprezentowali wykorzystanie maszyn wektorów nośnych kalibrowanych rozkładem Weibulla (W-SVM) do wykrywania włamań sieciowych. Analiza została wykonana także na zbiorze danych KDDCUP'99 i porównała działanie W-SVM z klasyczną wersją algorytmu maszyn wektorów nośnych (SVM) rozszerzoną o prawdopodobieństwo klasyfikacji bazujące na kalibracji Platta. Z badań autorów wynikło, że dokładność klasyfikacji oprócz scenariusza zamkniętozbiorowego była wyższa dla W-SVM.

Z kolei, w [117] Z. Zhang i in. zaproponowali system wykrywania włamań sieciowych umożliwiający działanie w scenariuszu otwartozbiorowym. Stworzony system składa się z trzech części. Pierwsza z nich bazuje na splotowej sieci neuronowej (CNN) i dokonuje klasyfikacji znanych ataków oraz rozpoznaje te, które są nieznane. Następnie, w części drugiej nieznane ataki są klastrowane, aby je dokładniej zidentyfikować. W końcu, w części trzeciej system wykorzystuje stworzone klastry, żeby zaktualizować klasyfikator o wiedzę na temat dotychczas nieznanych ataków.

Inne podejście do wykrywania nieznanych ataków sieciowych bazujące na splotowej sieci neuronowej (CNN) przedstawili w [116] Y. Zhang i in. Omawiana sieć przetwarza pliki zawierające zrzuty ruchu sieciowego (pcap) w postaci obrazów 40x40 pikseli, by w ostatniej warstwie bazującej na OpenMax dokonać klasyfikacji ruchu sieciowego, w tym ewentualnego rozpoznania klas nieznanych. Autorzy dokonali także analizy efektywności detekcji swojego systemu bazując na zbiorach danych CTU-13 oraz CICIDS2017. Porównanie z klasyczną siecią CNN bez warstwy OpenMax wskazało na wyższy poziom miary F1 oraz precyzji dla rozwiązania autorów.

Natomiast system wykrywania włamań sieciowych zaprezentowany przez M. Soltani i in. w [96], umożliwia aktualizację modelu uczenia maszynowego. Działanie systemu można podzielić

na cztery fazy. W pierwszej przeprowadzana jest klasyfikacja ruchu sieciowego przy wykorzystaniu algorytmu uczenia maszynowego w scenariuszu otwartozbiorowym. Następnie w drugiej fazie zbierane są informacje o nieznanych atakach i następuje ich grupowanie. Z kolei w trzecim etapie analityk dokonuje klasyfikacji danych zebranych w fazie drugiej, która ma służyć do aktualizacji modelu uczenia maszynowego w fazie czwartej. Jako algorytm otwartozbiorowy autorzy przetestowali cztery podejścia: Deep Open Classification (DOC [95]), DOC++ – autorską propozycję autorów modyfikującą funkcję straty oryginalnego DOC, model bazujący na OpenMax oraz AutoSVM, w którym autokoder stosowy (stacked autoencoder) służy do zmniejszenia wymiarowości cech wejściowych, przekazywanych do zestawu jednoklasowych maszyn wektorów nośnych. Autorzy dokonali także porównania efektywności klasyfikacji, z którego wynika, że DOC++ osiąga najlepsze wyniki.

Następnie, J. Yang i in. w [113] zaprezentowali system detekcji włamań sieciowych bazujący na autokoderze wariacyjnym (variational autoencoder). Proces detekcji składa się z dwóch części: w pierwszej dokonywana jest klasyfikacja znanych ataków, natomiast w drugiej identyfikacja ataków jeszcze nieznanych. W trakcie treningu podsystemu odpowiedzialnego za rozpoznanie nieznanych ataków badany jest błąd rekonstrukcji między danymi wejściowymi a wyjściem podsieci generatywnej tego podsystemu, który jest modelowany przy użyciu teorii wartości ekstremalnych. W trakcie predykcji model ten jest wykorzystywany do identyfikacji klas nieznanych. Autorzy dokonali także porównania wyników klasyfikacji ich systemu z modelami bazującymi na W-SVM, EVM oraz prostym modelu bazującym na zestawie dwóch podmodeli: lasów losowych badających przynależność do klas, oraz zestawu jednoklasowych maszyn wektorów nośnych badających czy klasa jest rzeczywiście znana. Wyniki wskazały, że podejście autorów jest efektywniejsze od innych modeli pod względem miary F1.

Powstały także systemy wykrywania włamań sieciowych operujące w scenariuszu otwartozbiorowym wyspecjalizowane do pracy w konkretnych środowiskach.

W pracy [105] C. Wang i in. przedstawili system wykrywania włamań sieciowych w przemysłowych systemach sterowania (Industrial Control Systems). Autorzy stworzyli sieć neuronową, w której zastosowali warstwę OpenMax w celu wykrycia klas nieznanych. Dodatkowo, na etapie treningu zastosowano centralną funkcję straty umożliwiającą lepsze modelowanie różnic między klasami.

Natomiast w [18] G. Bovenzi i in. zaprezentowali hierarchiczny system wykrywania włamań dla urządzeń IoT. Działanie systemu bazuje na dwóch fazach. W pierwszej, bazującej na głę-

bokim autokoderze, dokonuje się analizy ruchu sieciowego w celu prostej klasyfikacji na ruch normalny i nieprawidłowy. Jeśli uzyskany wynik wskazuje na anomalię, to dane przekazywane są do fazy drugiej, gdzie dokonywana jest klasyfikacja do jednej ze znanych klas lub odrzucenie jako nieznanego ataku. Natomiast w drugiej fazie wykorzystywany jest algorytm uczenia maszynowego (według analizy autorów najlepszy był algorytm lasów losowych) oraz prawdopodobieństwa detekcji klas. Jeśli maksymalna wartość tego prawdopodobieństwa spośród wszystkich klas nie przekroczy zdefiniowanego progu, atak uznawany jest za nieznany.

Problem klasyfikacji otwartozbiorowej był rozważany również w systemach detekcji wyspecjalizowanych do wykrywania konkretnych typów zagrożeń.

Y. Lai i in. w [62] przedstawili algorytm wykrywania nieznanych domen generowanych algorytmicznie (DGA) - OpenSMax. Klasyfikacja odbywa się w dwóch etapach. W pierwszym dokonywana jest klasyfikacja znanych klas DGA bazująca na długiej pamięci krótkotrwałej (LSTM) oraz warstwach w pełni połączonych, analizujących domenę najwyższego oraz drugiego poziomu. W drugim etapie detekcji rozpoznawane są nieznane klasy DGA. W tym celu dwie ostatnie warstwy sieci neuronowej pierwszego etapu (OpenMax oraz softmax) służą jako dane wejściowe dla zbioru jednoklasowych maszyn wektorów nośnych (One-Class SVM), które wykorzystywane są do rozpoznawania nieznanych klas DGA. Do oceny eksperymentalnej zaproponowanego rozwiązania wykorzystano zestawy danych Alexa-1M oraz z systemu detekcji DGA firmy 360. Natomiast do celów porównawczych użyto sieci splotowe bazujące na modelach OpenMax oraz DOC [95].

Z kolei C.-S. Shieh i in. zaprezentowali w [94] system do detekcji ataków DDoS bazujący na dwukierunkowych długich pamięciach krótkotrwałych (Bi-Directional Long Short-Term Memory - BI-LSTM) oraz modelu mieszanin gaussowskich (Gaussian Mixture Model - GMM). W pierwszym etapie detekcji sieć Bi-LSTM dokonuje podziału ruchu sieciowego na ruch regularny oraz ataki DDoS. Następnie dokonywana jest detekcja nieznanego ruchu sieciowego używając modelu mieszanin gaussowskich. W przypadku wykrycia takiego ruchu jest on przekazywany do analityka w celu określenia jego charakteru. Informacja ta jest następnie stosowana do aktualizacji modelu w sieci Bi-LSTM. W badaniach efektywności rozwiązania używane są dwa zbiory danych: CIC-IDS2017 oraz CICDDoS2019.

Scenariusz otwartozbiorowy wykorzystywany jest także do klasyfikacji plików binarnych złośliwego oprogramowania.

W [48] M. Hassen i P. K. Chan zaprezentowali model klasyfikacji plików binarnych ba-

zujący na splotowej sieci neuronowej. Z badanych danych wyodrębnili grafy wywołań funkcji (function call graph), które były danymi wejściowymi badanych sieci neuronowych. Autorzy zaproponowali nową funkcję straty, która maksymalizuje separację między klasami i minimalizuje odległość między próbkami a średnimi ich klas. Identyfikacja klas nieznanych odbywa się poprzez obliczenie miary odstawania, definiowanej jako odległość badanej próbki od średniej dla każdej klasy znanej. Jeśli odległość jest większa niż zdefiniowany próg, próbka uznawana jest za należącą do klasy nieznanej.

Z kolei J. Guo i in. w [45] zaprezentowali system klasyfikacji plików binarnych złośliwego oprogramowania. System wykorzystuje generatywną sieć przeciwstawną do tworzenia syntetycznych próbek, które są przekazywane do klasyfikatora wraz z rzeczywistymi próbkami. Według autorów taka architektura pozwala na zwiększenie efektywności klasyfikacji ze względu na wzajemny dodatni wpływ sieci generatywnej i klasyfikatora na siebie.

Innym interesującym obszarem badań wykorzystującym scenariusze otwartozbiorowe jest klasyfikacja ruchu sieciowego pod względem aplikacji generującej dany ruch. Podejścia takie nie są skupione na wykrywaniu ruchu sieciowego złośliwego oprogramowania, lecz na identyfikacji konkretnych, różnych aplikacji. Nie mniej rozwiązania stosowane w takich systemach stanowią dobre źródło inspiracji do tworzenia systemów wykrywania złośliwego oprogramowania.

W pracy [114] L. Yang i in. zaprezentowali porównanie różnych modeli uczenia maszynowego do klasyfikacji ruchu sieciowego, umożliwiających detekcję nieznanych aplikacji. Proces porównania został podzielony na dwie części. W pierwszej porównano klasyfikatory rozpoznające klasy znane bazujące na splotowej sieci neuronowej oraz na algorytmie XGBoost. W drugiej części porównano modele umożliwiające detekcję klas nieznanych bazujące na kilku metodach: klastrowaniu danych wejściowych wykorzystującej metodę k-średnich (k-means), klastrowaniu wektora cech wejściowych w przypadku sieci neuronowej, analizie wyjścia bazującej na SoftMax i OpenMax, a także modelu propagacji wstecznej zaproponowanej przez autorów.

J. Li i in. w [65] zaproponowali system identyfikacji nieznanego zaszyfrowanego ruchu sieciowego. Do tego celu wykorzystali model syjamskich sieci neuronowych, który został zmodyfikowany w celu zmniejszenia zużycia zasobów m.in. poprzez uproszczenie wektorów cech wejściowych oraz architektury sieci. System został porównany do niemodyfikowanych syjamskich sieci neuronowych oraz jednoklasowej maszyny wektorów nośnych, dając porównywalne wyniki efektywności klasyfikacji, ale skracając czas treningu oraz testów.

Tabela 3.3 przedstawia zestawienie i porównanie artykułów, których przegląd został za-

mieszczony powyżej.

Artykuł	Rok publikacji	Obszar działania	Algorytm uczenia maszynowego	Rozwiązanie otwartozbiorowe	Dane wejściowe
J. Henrydoss i in. [50]	2017	IDS	EVM	EVM	przepływ
S. Cruz i in. [22]	2017	IDS	W-SVM	W-SVM	przepływ
Z. Zhang i in. [117]	2021	IDS	CNN + klastrowanie	OpenMax	przepływ
Y. Zhang i in. [116]	2020	IDS	CNN	OpenMax	przepływ + przesyłane dane
M. Soltani i in. [96]	2021	IDS	CNN	DOC, OpenMax	przepływ + przesyłane dane
J. Yang i in. [113]	2021	IDS	VAE	VAE + model EVT	przepływ
C. Wang i in. [105]	2021	IDS – ICS	ANN	OpenMax	przepływ
G. Bovenzi i in. [18]	2020	IDS- IoT	AE + RF	RF + próg odrzucenia	przepływ + przesyłane dane
Y. Lai i in. [62]	2020	DGA	LSTM	OpenMax i softmax + zestaw OC-SVM	Nazwa domenowa
C.-S. Shieh i in. [94]	2021	DDoS	Bi-LSTM	GMM	przepływ
M. Hassen i in. [48]	2020	Pliki binarne	CNN	Modyfikacja funkcji straty oraz funkcja odległości od klas	Grafy wywołań funkcji
J. Guo i in. [45]	2021	Pliki binarne	GAN + CNN	Rozkład prawdopodobieństwa oraz próg odrzucenia	Zależny od źródła danych, m.in. bitmapy
L. Yang i in. [114]	2021	Klasyfikacja ruchu sieciowego	CNN	Autorska propagacja wsteczna	przepływ
J. Li i in. [65]	2021	Klasyfikacja ruchu sieciowego	SEEN	SEEN	przepływ + przesyłane dane

Tabela 3.3: Porównanie artykułów dotyczących klasyfikacji ruchu sieciowego lub złośliwego oprogramowania w scenariuszu otwartozbiorowym.

W rozdziale 6 rozprawy zaprezentowano podejście dokonujące klasyfikacji rodzin złośliwego oprogramowania przy użyciu protokołu HTTP w scenariuszu otwartozbiorowym, co według wiedzy autora niniejszej rozprawy nie było podejmowane wcześniej w literaturze. Różni się ono od rozwiązań przedstawionych w zaprezentowanym powyżej przeglądzie literatury. Przedstawione systemy wykrywania włamań sieciowych skupiają się na wykryciu nieznanych ataków, nie tylko wywoływanych przez złośliwe oprogramowanie, w większości bazując na informacjach pozyskanych z tzw. przepływów sieciowych (flows) i przesyłanych w nich danych, abstrahując od konkretnych protokołów. Podejście prezentowane w rozprawie koncentruje się natomiast ściśle na problemie klasyfikacji złośliwego oprogramowania przy użyciu danych dostępnych w

żądaniach protokołu HTTP. W porównaniu z zaproponowanym rozwiązaniem przedstawione w literaturze systemy wyspecjalizowane są w wykrywaniu konkretnych zagrożeń (DGA lub DDoS) oraz dostosowane do analizy odmiennych rodzajów danych i identyfikacji konkretnych zachowań. Pozostałe metody zaprezentowane powyżej albo dokonują klasyfikacji złośliwego oprogramowania na innym typie danych wejściowych, czyli plikach binarnych lub artefaktach z nimi powiązanymi, albo specjalizują się w klasyfikacji ruchu sieciowego należącego do aplikacji niezłośliwych.

4 Analiza ruchu sieciowego protokołu HTTP złośliwego oprogramowania

Protokół HTTP może być używany zarówno przez złośliwe oprogramowanie, jak i przez aplikacje niezłośliwe. Odróżnienie tych dwóch rodzajów ruchu sieciowego nie jest możliwe bez poznania ich cech charakterystycznych. Przyjmując perspektywę niniejszej rozprawy, skupiającej się na żądaniach protokołu HTTP, można odnaleźć badania zajmujące się prezentacją tych charakterystycznych cech. Przykładem może być Rossow i in. [87] opisujący analizę ruchu sieciowego ponad 100 tys. próbek złośliwego oprogramowania, Calzarossa i Massari w [19] opisujący analizę ruchu sieciowego protokołu HTTP przychodzącego do serwera ich uczelni, czy Montoro [76] opisujący system punktacji różnych cech żądań w celu określenia ich złośliwości. Niestety, dotychczasowe analizy posiadają kilka wad takich jak mała liczba analizowanych cech żądań HTTP, ograniczona liczba analizowanych rodzin złośliwego oprogramowania lub próbek, skupienie się na prezentacji systemów detekcji bez odpowiedniej analizy cech lub brak formalnego, akademickiego dowodu potwierdzającego użyteczność wskazywanych cech.

Żeby rozwiązać powyższe problemy, przeprowadzono badania nad charakterystyką ruchu sieciowego protokołu HTTP złośliwego oprogramowania. Celem badań było odnalezienie cech żądań HTTP, które byłyby charakterystyczne dla złośliwego oprogramowania, tym samym umożliwiając odróżnienie go od aplikacji niezłośliwych. Aby to osiągnąć, dokonano analizy porównawczej ruchu sieciowego tych dwóch typów aplikacji pod kątem zdefiniowanego zestawu cech.

Głównym wkładem pracy badawczej prezentowanej w tym rozdziale jest:

- Przeprowadzenie przeglądu cech żądań HTTP dotychczas używanych do wykrywania złośliwego oprogramowania w źródłach akademickich.
- Przeprowadzenie akademickiej weryfikacji użyteczności cech żądań HTTP używanych w dostępnych publicznie źródłach nieakademickich (jak np. moduły systemu Cuckoo Sandbox [23], [24]).
- Zaproponowanie zestawu cech żądań HTTP, które potencjalnie mogą być użyte do rozpoznawania złośliwego oprogramowania, w tym także propozycja nowych, dotychczas nieopisanych cech.
- Analiza i identyfikacja cech żądań HTTP złośliwego oprogramowania umożliwiających

odróżnienie takich żądań od żądań aplikacji niezłośliwych, a także przeprowadzenie analizy ich wzajemnego wpływu.

- Przedstawienie listy cech żądań HTTP złośliwego oprogramowania umożliwiających odróżnienie takich żądań od żądań aplikacji niezłośliwych, wsparte przykładami jej praktycznego użycia.

W badaniach analizowano trzy grupy cech żądań HTTP: cechy struktury żądania, cechy wartości nagłówków żądania oraz cechy danych przesyłanych w żądaniu (tzw. payload). Cechy zostały zdefiniowane na bazie doświadczenia w analizie złośliwego oprogramowania autora niniejszej rozprawy, będącego specjalistą bezpieczeństwa w zespole CERT Polska, oraz wcześniejszych pracach znanych z literatury światowej, co prezentuje Tabela 4.1.

Grupa cech	Źródło badań
Struktura żądania	Montoro [76], Cuckoo [23], [24], Calzarossa i in. [19], Rossow i in. [87], Nelson [77], K. Li i in. [67]
Wartości nagłówków żądania	Montoro [76], Lewis [64], Mizuno i in. [74], Calzarossa i in. [19], Z. Li i in. [69], Kheir [57], Rossow i in. [87], Nelson [77], K. Li i in. [67], Perdisci i in. [81]
Dane przesyłane w żądaniu	Perdisci i in. [81]

Tabela 4.1: Źródła badań na temat analizowanych grup cech żądań.

W grupie cech struktury żądania analizowane były następujące cechy[14]:

1. wersja protokołu,
2. metoda żądania,
3. powtórzenia nagłówków*,
4. brak dwukropka w polu nagłówka*,
5. liczba nagłówków w żądaniu,
6. częstość występowania nagłówków,
7. literówki w nazwach nagłówków,
8. obecność mechanizmu łączenia żądań (pipelining)*,

[14]gwiazdką oznaczone zostały cechy zaproponowane przez autora niniejszej rozprawy

9. port docelowy protokołu TCP żądania.

W grupie cech wartości nagłówków żądania analizowane były poniższe cechy[15]:

1. obecność znaku spacji jako pierwszego znaku w polu nagłówka,
2. obecność znaku spacji przed znacznikiem nowej linii CRLF,
3. obecność znaku spacji przed dwukropkiem, średnikiem lub przecinkiem,
4. obecność znacznika nowej linii innego niż CRLF*,
5. obecność podwójnego znaku spacji,
6. obecność niestandardowego znaku odstępu w polu nagłówka*,
7. obecność znaku spoza zestawu ASCII w wartości nagłówka*,
8. wartość nagłówka *Accept-Language*,
9. wartość nagłówka *Accept-Encoding*,
10. wartość nagłówka *Connection*,
11. wartość nagłówka *Host* header value*,
12. wartość nagłówka *User-Agent*.

Analiza wartości nagłówka *Host*, mimo że przeprowadzana już wcześniej (np. w Li i in. [69]), w prezentowanych badaniach była pogłębiona, by wskazać typ wartości, przykładowo, czy był to adres IP, domena lub inna wartość. Dlatego też została oznaczona jako nowy typ analizy na powyższej liście.

W grupie cech danych przesyłanych w żądaniu analizowane były[16]:

1. rozmiar pola przesyłanych danych,
2. wartość entropii przesyłanych danych,
3. obecność znaków spoza zestawu ASCII w przesyłanych danych*,
4. obecność żądań innych niż *POST* zawierających dane*,

[15]gwiazdką oznaczone zostały cechy zaproponowane przez autora niniejszej rozprawy

[16]gwiazdką oznaczone zostały cechy zaproponowane przez autora niniejszej rozprawy

5. obecność nagłówka *Referer* w żądaniach z metodą *POST*.

Badania zostały przeprowadzone z wykorzystaniem dwóch zbiorów danych ruchu sieciowego: złośliwego oprogramowania oraz popularnych przeglądarek internetowych, reprezentujących aplikacje niezłośliwe. Dane ruchu sieciowego złośliwych aplikacji pochodziły ze środowiska automatycznej analizy malware'u zespołu CERT Polska oraz projektu długoterminowej analizy złośliwego oprogramowania Malware Capture Facility Project (MCFP) [17]. Dane te zostały oznaczone przy użyciu systemu wykrywania włamań sieciowych Snort IDS zaopatrzonego w reguły ET Pro [18] oraz zarejestrowane reguły Snort [19].

Do celów statystycznych dane złośliwego oprogramowania zostały pogrupowane według alarmów wywoływanych przez żądania. Żądania wywołujące takie same zestawy alarmów były łączone w grupy, tworząc grupy żądań. W ten sposób grupy żądań tworzyły jednostki używane w obliczeniach statystycznych zamiast pojedynczych żądań. Taki sposób grupowania został przyjęty w celu zmniejszenia wpływu różnego rozkładu liczby żądań w rodzinach. Podział na grupy żądań nie został przyjęty w przypadku ruchu sieciowego aplikacji niezłośliwych. Dodatkowo, grupy żądań były rozpatrywane ze względu na kategorię złośliwego oprogramowania. 20 kategorii określało charakter działania analizowanych programów, przykładowo, wykradanie danych użytkowników, szyfrowanie pliku dla okupu, wysyłka spamu itd.

Analiza częstości występowania cech żądań HTTP wykazała, że niektóre z nich w wyraźny sposób umożliwiają odróżnienie złośliwego oprogramowania od aplikacji niezłośliwych. Te cechy to[20]:

- wersja *1.0* protokołu HTTP,
- od 0 do 3 nagłówków w żądaniu,
- port docelowy inny niż 80,
- brak któregokolwiek z nagłówków: *Accept*, *Accept-Encoding*, *Accept-Language*, *Referer*, *Connection*,
- brak nagłówka *User-Agent*,
- wystąpienie żądania z metodą *POST*, ale bez nagłówka *Referer*,

[17] `https://www.stratosphereips.org/datasets-malware`

[18] `https://www.proofpoint.com/us/threat-insight/et-pro-ruleset`

[19] `https://www.snort.org/downloads/#rule-downloads`

[20] gwiazdką zostały oznaczone cechy wskazane przez autora niniejszej rozprawy

- wartość nagłówka *Host* inna niż nazwa domenowa*,
- niestandardowa wartość nagłówka *User-Agent*,
- wystąpienie żądania z metodą *GET* zawierającego dane*,
- wysoka entropia przesyłanych danych,
- wystąpienie znaków spoza zestawu ASCII w przesyłanych danych*.

Na bazie powyższej listy cech dokonano analizy częstości występowania ich par. Dzięki temu możliwe było wskazanie, które cechy najczęściej występują ze sobą w żądaniach HTTP złośliwego oprogramowania:

- wraz z małą liczbą nagłówków występuje brak nagłówka *User-Agent*,
- żądania, które mają wysoką entropię, w nagłówku *Host* mają wartość inną niż nazwa domenowa lub są to żądania z metodą *POST*, ale nie zawierające nagłówka *Referer*,
- żądania wysyłane na port inny niż 80 zawierają niestandardową wartość nagłówka *User-Agent*,
- żądania z metodą *GET* oraz przesyłające dane mają małą liczbę nagłówków lub entropia przesyłanych przez nich danych jest wysoka, lub przesyłane dane zawierają znaki spoza zestawu ASCII,
- żądania z metodą *POST* niezawierające nagłówka *Referer* w przesyłanych danych mają znaki spoza zestawu ASCII,
- żądania, które nie zawierają nagłówka *Accept*, nie zawierają także nagłówka *Accept-Encoding* lub nagłówka *Accept-Language*,
- żądania niezawierające nagłówka *Connection* nie zawierają także nagłówka *Accept* lub nagłówka *Accept-Encoding*, lub nagłówka *Accept-Language*,
- żądania z portem docelowym innym niż 80 w nagłówku *Host* zawierają wartość inną niż nazwa domenowa,
- jeśli żądanie jest wersji *1.0* protokołu HTTP, to przy obecności metody *POST* nie występuje nagłówek *Referer*.

Przeprowadzone badania wykazały, że istnieje zestaw cech żądań HTTP złośliwego oprogramowania, które umożliwiają odróżnienie go od aplikacji niezłośliwych. Cechy te można wykorzystać w praktyce, przykładowo, przy ręcznej analizie ruchu sieciowego lub tworzeniu reguł dla systemów wykrywania włamań sieciowych.

Omówiona powyżej analiza charakteru ruchu sieciowego protokołu HTTP złośliwego oprogramowania umożliwiła sformułowanie i konstrukcję cech żądań HTTP, które zostały wykorzystane w kolejnym etapie pracy badawczej nad identyfikacją żądań złośliwego oprogramowania, zaprezentowanym w rozdziale 5.

Pełen opis badań nad charakterem ruchu sieciowego protokołu HTTP złośliwego oprogramowania został opublikowany pod tytułem *Characterizing Anomalies in Malware-Generated HTTP Traffic* w czasopiśmie *Security and Communication Networks*, vol. 2020, Article ID 8848863 [11].

Hindawi
Security and Communication Networks
Volume 2020, Article ID 8848863, 26 pages
https://doi.org/10.1155/2020/8848863

Research Article

Characterizing Anomalies in Malware-Generated HTTP Traffic

Piotr Białczak[1] **and Wojciech Mazurczyk**[2]

[1]*CERT Polska/Research and Academic Computer Network (NASK), Kolska 12, Warsaw 01-045, Poland*
[2]*Warsaw University of Technology, Nowowiejska 15/19, Warsaw 00-665, Poland*

Correspondence should be addressed to Piotr Białczak; piotr.bialczak@cert.pl

Received 14 April 2020; Revised 18 June 2020; Accepted 7 August 2020; Published 1 September 2020

Academic Editor: Clemente Galdi

Copyright © 2020 Piotr Białczak and Wojciech Mazurczyk. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Currently, we are witnessing a significant rise in various types of malware, which has an impact not only on companies, institutions, and individuals, but also on entire countries and societies. Malicious software developers try to devise increasingly sophisticated ways to perform nefarious actions. In consequence, the security community is under pressure to develop more effective defensive solutions and to continuously improve them. To accomplish this, the defenders must understand and be able to recognize the threat when it appears. That is why, in this paper, a large dataset of recent real-life malware samples was used to identify anomalies in the HTTP traffic produced by the malicious software. The authors analyzed malware-generated HTTP requests, as well as benign traffic of the popular web browsers, using 3 groups of features related to the structure of requests, header field values, and payload characteristics. It was observed that certain attributes of the HTTP traffic can serve as an indicator of malicious actions, including lack of some popular HTTP headers and their values or usage of the protocol features in an uncommon way. The findings of this paper can be conveniently incorporated into the existing detection systems and network traffic forensic tools, making it easier to spot and eliminate potential threats.

1. Introduction

In the present-day Internet, one of the most commonly used protocols is the Hypertext Transfer Protocol (HTTP) [1, 2]. Its utilization is widespread as it is an essential component of web browsing. It also serves as a "backbone" of many services, even those standardized with other network protocols like e-mail and instant messaging. However, HTTP protocol prevalence is steadily decreasing in favor of TLS, HTTP/2, and FB-ZERO protocols, according to [1]. Deployment of an HTTP server is easy even for those who are not tech-savvy users, with many tutorials available in national languages. It is also often provided as a service by webhosting companies. On top of this, there is a lack of monitoring or blocking in many networks and easily achievable blending with legitimate network traffic. It is not surprising that malware developers use HTTP as a primary protocol to enable malicious communication. For example, according to Miller and Smith [3], HTTP is the most popular protocol used in C&C traffic, surpassing HTTPS. All this led the authors of this paper to focus on analyzing the HTTP protocol solely.

HTTP is used by malware for various purposes, for example, for connecting with the Command and Control (C&C) server to register/download commands, checking the external IP address of the infected host, and downloading additional modules. It is also used to perform DDoS (Distributed Denial of Service) attacks or create revenue by clicking on referral links. Such communication is masked by benign HTTP traffic which can be vastly different, depending on the application and its usage purpose. It must be noted that the HTTP protocol can be used by applications other than web browsers, for example, updaters, operating system mechanisms, application shops, and messengers. The main difference between the network traffic of such applications and the network traffic of web browsers lays in the characteristic of used addresses. The latter traffic can be potentially directed to any address, while in the former, the addresses are constant: they are either a set of domain names

or an IPs range. For example, addresses of servers used by Windows telemetry services or Windows update mechanisms are widely known and are listed in many manuals focusing on blocking these services with network firewalls [4, 5] or dedicated tools such as WindowsSpyBlocker (https://github.com/crazy-max/WindowsSpyBlocker/). Network traffic of these applications can be easily identified using, for example, publicly available address lists or a short analysis of the traffic in the network proxy log. Considering the above, the authors decided to focus only on the web browser traffic as the other popular HTTP-based applications are relatively easy to be identified and filtered out from the network traffic.

The analysis of HTTP traffic characteristics presented in the current malware behavior research [6–9] suggests that some malware families' HTTP requests differ from those generated by benign applications. This is especially visible when compared to the network traffic of applications operated by humans, e.g., web browsers. However, to the authors' best knowledge, there is no extensive study which systematically identifies and analyzes dissimilarities between the malicious (malware) and benign (web browsers) HTTP traffic.

To fill this gap, the authors have thoroughly analyzed HTTP requests of both malware and browser traffic (using recent traffic sources), in order to establish their distinctive features. The research has focused solely on the Microsoft Windows operating systems family, as it is still the most frequently attacked platform—in 2018, more than half of the newly developed malware targeted these systems [10]. The conducted investigation explores a set of features and was created based on the authors' own experience with real malware samples' analyses and previous research work in this area (see Section 2). The chosen features reflect the structure of requests, values of different HTTP protocol fields, and the analysis of payload data. The main objective is to identify which parts of HTTP requests are different in malware and the browser network traffic and which can be identified as general features for distinguishing between these two types. The features and their values deviating from standards defined by network traffic originating from browsers will be defined as anomalies. Some of the analyzed features can be seen as anomalies because they do not conform to standards or registered values, and they are present in both malware and browser network traffic. In such cases, the frequency of such occurrences will be quantified.

The main motivation behind this work is to provide other researchers with a list of identified anomalies of malware HTTP traffic. Such a list can be used directly by analysts when analyzing network traffic (e.g., during digital investigation) but also as an entry point for the design of malware detection systems. Availability of a well-described set of network anomalies can also help in developing other monitoring systems, for example, malware fingerprinting solutions. Therefore, the authors believe that this work will help fighting malicious software.

Considering the above, the main contributions of this paper are

(i) Conducting a survey of HTTP requests' features previously used to detect malware in the existing research and performing an academic verification of usefulness of features proposed by previous non-academic work

(ii) Proposing an improved set of HTTP requests' features, including original ones, which can be potentially utilized for malware identification

(iii) Identifying and analyzing malware HTTP requests' distinctive features and performing analysis of their influence on each other

(iv) Providing a list of malware HTTP requests' distinctive features, along with practical usage scenarios

The contributions of this paper in a summarized and concise form are presented in Sections 6.1 and 6.2.

Due to a substantial number of performed analyses, not all of them were described in this paper, in order to maintain its clarity. Included are only those results which can help distinguish between malware and HTTP network traffic.

The rest of the paper is structured as follows. Section 2 describes the existing work related to the HTTP-based anomaly detection. Section 3 explains the fundamentals of the HTTP protocol. In Section 4, an experimental methodology used in this paper is outlined in detail. Section 5 presents obtained experimental results. Section 6 investigates how our discoveries can be applied in practice to the existing detection solutions. In Section 7, several limitations of this work are discussed. Finally, Section 8 concludes this paper and outlines future work.

2. Related Work

This section reviews the existing works which are most closely related to the research conducted in this paper. To start with, academic research papers exploring the behavior of malware are described, as they can be directly compared with the below work. Several nonacademic sources are also investigated; they show or use features for identification of the malware HTTP requests.

Rossow et al. presented in [11] the results of analysis of malware network traffic. They analyzed more than 100,000 samples, from which about 43.8% performed network activity. The authors provide observations about DNS and HTTP traffic, but only the latter will be summarized here. Analysis of the HTTP requests revealed that 89.5% of samples sent *GET* and 56.3% sent *POST* requests. Furthermore, 144 unique header names were observed. 98.6% of samples specified the *User-Agent* header; however, only 31% of samples included correct values. Additionally, 50.6% of samples changed this value during execution. 44.3% of samples included the *Accept-Language* header; however, 24.1% of them did not respect the operating system language locale.

In [12], Nelson presented a framework for analyzing and visualizing malware network traffic. The framework expands on the Sandnet framework [11] and its analyses. It provides a

means for execution of malware samples and capturing their network traffic; it also provides analysis and clustering of protocols and visualization of the obtained results. To evaluate the framework, an analysis was conducted, providing manual inspection of 5 malware families and semi-automatic inspection of the whole dataset of 16,967 pcap files. In the latter part, the author analyzed network protocol breakdown and characteristics of DNS and HTTP protocols. Analysis of 118,035 HTTP requests included in the dataset was performed on multiple features, such as the request method, URI, or popular header values. The results reveal that in 86.7% of the captured files, *GET* requests were present. Also, in 86.8% of these files, *POST* requests were present. 24 unique header names were observed, as well as 33 unique *User-Agent* header values. The author also performed an analysis of *Accept-Language* and *Content-Type* headers, stating that their values can be used to identify malicious network traffic.

Calzarossa and Massari in [13] presented an evaluation of headers' usage in HTTP traffic. The authors monitored network traffic generated towards web servers at their university, focusing mainly on capturing HTTP requests. The analyzed a dataset which consisted of 315,000 requests, sent by about 6100 clients. The results indicated interesting characteristics of HTTP traffic. About 4% of requests were sent using HTTP/1.0, and the number of header fields was distributed between 0 and 14 (with mean 6.34). The number of unique header names was about 60, but the number of occurrences was different. *Host* and *User-Agent* headers were the most popular ones and appeared in more than 99% of requests, followed by *Connection*, *Accept*, and *From*. The authors also analyzed headers' usage patterns, i.e., popularity of headers' sets among requests. The 10 most popular patterns occurred in 81% of requests, and about two-thirds of requests shared one pattern. The authors also observed that the number of headers and usage patterns differed between the browsers and web robots, thus allowing to distinguish them easily.

As already mentioned, some nonacademic sources related to this research are presented below.

In a presentation "HTTP Header Hunter–Looking for Malicious Behavior into Your HTTP Header Traffic," Montoro presents the scoring system for the HTTP request headers [14]. The system inspects HTTP requests' features, whitelists and blacklists of the *User-Agent* header values, or top-level domains and the third-party data sources such as geoIP. The analyzed HTTP requests' features include presence of common headers (for example, *Cookie*, *Accept-Encoding*, and *Connection*), number of header fields in a request, protocol version, *User-Agent* header values' size, type of files being requested in URI, and presence of the *Host* header in HTTP/1.0 requests. He also proposed the usage of headers' ordering, response headers, and parameter names; however, this was not implemented in his work. The author's analysis showed that the malware sometimes does not include *User-Agent* or its value length is usually shorter than 90 bytes. Also, malware tends to send 1–3 headers in requests, and nonmalicious applications usually send more than 9 headers. The presented system adds a score to the features to provide information about maliciousness of requests, and it was tested on 6127 data streams. The resulting detection rate of 89.1% and a false-positive rate of 9.15% have been achieved.

Cuckoo malware sandbox system (https://cuckoosandbox.org/) provides community modules which analyze HTTP protocol traffic. The *network_cnc_http* module [15] provides information about "suspicious features which may be indicative of malware-related traffic." It analyzes the lack of the *Referer* header in the POST request, the lack of the *User-Agent* header in the *POST* and *GET* requests, the presence of HTTP 1.0 version requests, and the presence of the IP address in the *Host* header. The *multiple_useragent* module [16] verifies whether multiple *User-Agent* header values are used.

Lewis presented a paper about HTTP headers' heuristics for malware detection [17]. The author proposes utilization of some particular anomalies to help in the malware recognition. These include observing the *User-Agent* string for values which are nonstandard and different from the usual for the particular network, typographic errors in headers' names and values (additional whitespaces and misspellings of header names), and complexity of the requested resource, e.g., the length of the requested URL.

It must be noted that a large portion of academic research papers focus on describing malware detection systems using the HTTP protocol. A selected representation has been described below.

Mizuno et al. presented in [18] the malware detection system called BotDetector, which uses HTTP requests' header patterns. The system creates HTTP templates based on header fields; it does not focus on chosen fields, but on all of them. Each field is split into words, which are then evaluated using conditional probability of their appearance in a particular position of the header field. After performing calculations, header fields are clustered using the DBSCAN algorithm, thus producing the HTTP request template.

Li et al. presented a framework for detection and classification of network traffic of malicious Android application [19]. It is based on analysis of the HTTP protocol and organized into 3 components: training module, clustering module, and malware classification and detection module. Training module uses 5 features for model building. It includes values of the headers: *Host*, *Referer*, *User-Agent*, and *Content-Type* and the value of the request URI. This module uses the scoring mechanism which incorporates the header value occurrence frequency and its previous presence in the database.

Kheir in [20] introduced the malware taxonomy based on the *User-Agent* header values, which is used for detecting anomalous values proposed by the author. The *User-Agent* header values are clustered in a two-step process. During the first phase, they are clustered based on high-level features like length of the string and different character type frequencies. In the second step, the values are fine-grained and clustered based on the similarity of value parts. Finally, clusters are tokenized to produce HTTP signatures, used for detection purposes.

Li et al. in [21] presented the detection system for malware traffic. The system uses HTTP requests' features such as character distribution and the length of the URL, values of *Content-Type* and *User-Agent* headers, and ordering of the header in request.

In [22], the authors presented the malware detection system utilizing analysis of multiple HTTP requests to create behavior models. Statistical models were created for coarse-grained clustering, based on multiple requests of malware using, for example, the average length of the payload, response, or URI. For fine-grained clustering, they used the request method and lexical features of the URI.

To the authors' best knowledge, the papers presented above have the following disadvantages when compared with the research presented in this paper:

(i) The number of malware families, samples, or analyzed HTTP requests is smaller than that in this analysis

(ii) The scope and the number of analyzed features are limited

(iii) The existing work focuses on presentation of the detection system, without thoroughly (or only to the limited extent) exploring feature analysis

(iv) The identification of anomalies is not proved within the existing analyses

(v) Features of HTTP requests identified by nonacademic sources were not verified academically

Considering the above, the below work aims at filling these gaps by analyzing more extensively the dataset and providing systematic analysis of a large number of HTTP requests' features.

Some academic sources provide different approaches to the problem of malware and browser distinctiveness or to a broader problem of detection of malicious behavior. Two examples are presented below, along with discussion about connection to this paper.

Mimura and Tanaka in [23] presented a generic attack detection method based on proxy server logs and URL. The method is independent of attack methods and does not require designing features for classifiers. The authors used the paragraph vector algorithm to capture the context between multiple lines of proxy logs and produce vectors for three classifiers: support vector machine, random forests, and multilayer perceptron. The experimental results proved that the method can detect unseen drive-by-download attacks and C&C traffic in proxy server logs. Mimura et al. in their paper focused on detection of malware behavior, while in this paper, authors emphasize on the search of particular features, which distinguish malware and browser traffic. Moreover, the method used by Mimura et al. does not require designing of features and can be seen as independent of particular features, whereas in this paper, the authors provided a static list of features, which are analyzed.

Nia et al. in [24] presented a detection method of new generations of cyber threats using the pattern-based random walk. The authors proposed to use a limited method of random walk called the self-avoiding walk, in order to create a behavioral graph based on network traffic. The method uses the ordered triple of *time*, *size*, and *direction*, created for packets in analyzed network flows. The authors created a database of behavioral graphs for known threats. If the analyzed packet set has a similar graph created by the self-avoiding walk algorithm, then it is detected as malicious. The authors reported a true detection rate of 95% for malicious traffic. Nia et al. focused on detection of threats, while authors of this paper emphasize on identification of features distinguishing malware and browser traffic. Moreover, the method by Nia et al. works on flow-level features and is independent of higher network-level protocols, while authors of this paper focused on specific parts of the HTTP protocol, which is an application-level protocol.

3. HTTP Protocol Basics

HTTP protocol in version 1.1 was originally defined in RFC 2616 [25] in June 1999. The RFC has been obsoleted by RFCs 7230–7235 [26–31]. Two earlier versions of the protocol exists: 0.9 and 1.0, where the latter one was defined in RFC 1945 [32]. However, only 1.0 version should still be supported.

The HTTP protocol is based on the client-server architecture, where the client sends a request and the server replies to this request with a response. Request methods defined by RFC 7231 [27] are presented below with short descriptions:

(i) *GET*: the primary method for resource retrieval; it usually does not carry payload data, but this is not forbidden

(ii) *HEAD*: this method is similar to *GET*, but the server must not send any data in the response body (except for the header section)

(iii) *POST*: the method of signalling to the server request for processing data enclosed in the payload

(iv) *PUT*: this method is used to create or replace the state of the target resource with the state enclosed in the message payload

(v) *DELETE*: according to RFC 7231, this method "requests that the origin server removes the association between the target resource and its current functionality"

(vi) *CONNECT*: this method is used to signal proxy request for creation of a connection with a destination server provided in the request

(vii) *OPTIONS*: this method is used for discovering information about the communication options available for the requested resource

(viii) *TRACE*: this method is used to request the server to resend the request back to the client; it must not contain payload data

An example of an HTTP *GET* request is presented in Figure 1.

```
GET / HTTP/1.1
Host: cert.pl
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64;
            rv:63.0) Gecko/20100101 Firefox/63.0
Accept: text/html,application/xhtml+xml,
        application/xml;q=0.9,*/*;q=0.8
Accept-Language: pl,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Figure 1: Listing of an exemplary HTTP request (some of the lines were wrapped to fit the table).

The first line in the request in Figure 1 indicates the request method, "GET," requested Uniform Resource Identifier (URI)—"/," and the protocol statement with protocol version—"HTTP/1.1." At the end, the Carriage Return Line Feed (CRLF) is added. Further lines contain header fields, each with a field name and a field value, separated by a colon ":". Field names are case insensitive, and field values consist of printable US-ASCII characters. In practice, the majority of the HTTP client implementations follow such behavior and use printable US-ASCII characters in the header fields. Additionally, RFC 7230 obsoleted the usage of non-US-ASCII characters in these fields.

According to RFC 7230, there cannot be any space or horizontal tabulator between the field name and the colon. There can however be any number of such characters between the colon and the field value, as well as between the field value and the end of the header field. Usually, there is only one space before the field value and no whitespace characters at the end of the field. Additionally, the header field order does not have any special meaning.

According to Section 5 of RFC 7231, the reason behind the usage of header fields is "[...] to provide more information about the request context, make the request conditional based on the target resource state, suggest preferred formats for the response, supply authentication credentials, or modify the expected request processing." Moreover, the header fields are "ought to be registered with IANA" (according to Section 3.2.1. of RFC 7230). The registry can be accessed at https://www.iana.org/assignments/message-headers/message-headers.xhtml.

Despite a rather extensive number of registered header field names (or shortly, headers), some of them are more popular than others. Many of these fields are included in this analysis. They are listed below:

(i) *Host*—carries information about the host, port, and target URI; this header field must be present in all requests of the HTTP protocol version 1.1

(ii) *Accept*—specifies response media types that are acceptable by the client

(iii) *Accept-Language*—characterizes the set of natural languages preferred by the client in the response

(iv) *Accept-Encoding*—depicts acceptable content codings

(v) *User-Agent*—indicates which application is the source of the request

(vi) *Connection*—specifies control options of the connection desired by the client

(vii) *Referer*—points to the source URI from which requested URI originates

Once the set of header fields is in place, an additional CRLF tag is inserted. At this point, a message body can be added. The message body may end with the CRLF.

The message body can contain encoded data if the original payload is compressed. The popular methods used for this purpose are *deflate* and *gzip*. Additionally, data can be divided and encoded with chunked transfer coding. In such a way, parts of the data are sent using chunk-size information.

RFC 7230 also defines pipelining mechanism. When using this communication mode, the client can send multiple requests without waiting for corresponding responses.

4. HTTP Traffic Analysis Overview and Experimental Methodology

4.1. HTTP Traffic Analysis Overview. An overview of the malware HTTP requests' analysis workflow is presented in Figure 2. The process of analysis begins with choosing pcap files which contain HTTP request traffic from datasets. The files are filtered with the tshark (https://www.wireshark.org/docs/man-pages/tshark.html) filter so that only the TCP protocol segments containing HTTP requests that are not OCSP (Online Certificate Status Protocol) requests remain. Files with HTTP requests are then fed to an IDS system. In the proposed approach, Snort IDS with ET Pro rules (https://www.proofpoint.com/us/threat-insight/et-pro-ruleset) and Snort registered rules (https://www.snort.org/downloads/#rule-downloads) are used to check for alert logs triggered by the network traffic within the pcap files. Labeling of HTTP requests begins with semimanual check of the generated alerts. They are reviewed in order to filter out those which do not present HTTP traffic, or without information about maliciousness, or alerting about nonmalicious applications or services, for example, Tor traffic. The top 10 most common Snort IDs (SIDs) with alert messages after semimanual filtering are presented in Table 1. Alerts for Trojan Dridex and ransomware Locky are frequent in the dataset, and this impact will be discussed in Section 4.2. In the next step of request labeling, every unique SID is labeled manually with the malware family name, depending on the name provided by the alert message. If no family name is present, it is labeled as *No-name*. If different variants of family names are present (occurring when various vendors/malware analysts provide different names), they are normalized to one name. In the final step of HTTP request labeling, every request is assigned with a set of Snort IDs alerted for a particular request. The assignment is done automatically, on the basis of correlation of tuple: *source IP address, source port number, destination IP address, destination port number,* and *timestamp* between the tshark output for every request and the corresponding tuple in the IDS alert set. Please note that the *timestamp* values are transformed and normalized, in order to prevent any time

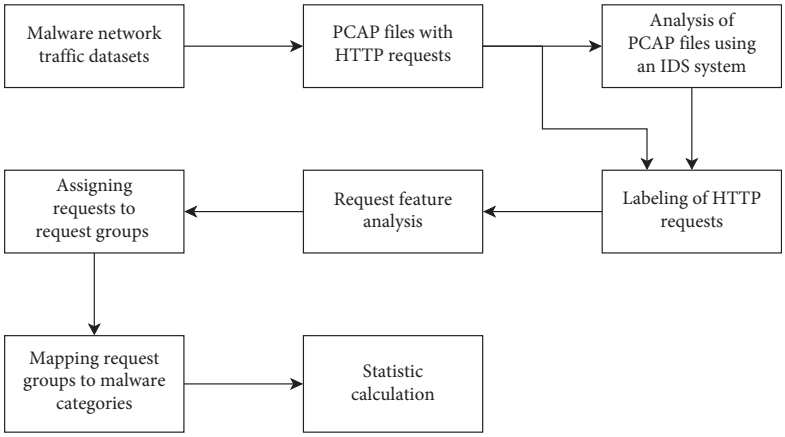


Figure 2: An overview of the malware HTTP requests' analysis workflow used in the paper.

Table 1: The top 10 most common Snort IDs and alert messages observed in the analyzed traffic.

Snort ID	Alert message
1:43685:1	MALWARE-OTHER Win.Trojan.Nemucod variant outbound connection
1:2023577:1	ET TROJAN Locky CnC Checkin HTTP Pattern
1:32678:2	MALWARE-CNC Win.Trojan.Dridex variant outbound connection
1:33145:2	MALWARE-CNC Win.Trojan.Dridex initial outbound connection
1:2019478:1	ET TROJAN Dridex POST Checkin
1:2023551:1	ET TROJAN Locky CnC checkin Nov 21
1:2023552:1	ET TROJAN Locky CnC checkin Nov 21 M2
1:2807610:2	ETPRO TROJAN DirtJumper DDoS (INBOUND)
1:2016879:2	ET POLICY Unsupported/Fake Windows NT Version 5.0
1:2821731:3	ETPRO CURRENT_EVENTS MalDoc Request for Payload Aug 17, 2016

deviations, if the request timestamp reported by tshark is different than that reported by the IDS. If the tuples are the same, the request is assigned with a particular SID of the alert, along with the malware family name. In the case of multiple SIDs assigned to one request, but with different family names, the request is analyzed manually to provide the final label.

HTTP requests labeled as malicious are evaluated using feature analyzers. The feature list is static and is discussed in Section 4.3. The analyzers utilize popular tools to perform the actual analysis of the features. Feature extraction and analysis process can be divided into three steps: (i) analysis of basic features, (ii) analysis of complex features, and (iii) analysis of payload features.

Basic feature extraction and analysis covers all features which can be analyzed by their direct value extracted from the HTTP request, for example, version of the protocol, the type of the request method, or the popular header values. For the extraction process, tshark is used to provide values of particular fields, supported by the tool, for example, the *http.request.method* for extraction of the request method.

Extraction and analysis of complex features covers the process of obtaining values for features which are not direct values of request fields but involves further analysis of such fields. These features include, for example, verifying the presence of unusual whitespace characters or non-US-ASCII characters in the header values. The process is performed by using the Scapy Python library (https://scapy.net/) for analysis of pcap files and extraction of HTTP requests. The actual analysis of data provided by Scapy is continued using Python scripts, depending on the particular feature.

The final step of feature extraction and analysis is performed for requests which have payload. Firstly, tshark Lua scripts are used for extraction of the payload data. Then, a Perl script is used to detect the presence of the non-US-ASCII characters in the extracted payload. Finally, the payload entropy is calculated using *ent*—a pseudorandom number sequence test program (http://www.fourmilab.ch/random/).

After analyzing request features, the requests are assigned to request groups and malware categories in order to prepare data for statistic calculation. This part of the process is extensively described in Section 4.2.

It must be noted that the benign browser-based HTTP traffic was directly analyzed using the same set of analyzers as described above. Additionally, it was also fed into the IDS system in search for any traces of malicious traffic. The results did not show any significant alerts.

4.2. HTTP Traffic Statistic Calculation. The traffic statistics for the malicious dataset rely on grouping analyzed requests into different sets of requests, called request groups. Requests form a request group when they trigger the same

alerts at the IDS. Exemplary relations between inspected HTTP requests and request groups are presented in Figure 3.

The mechanism of assigning HTTP requests to particular request groups is as follows. For all HTTP requests reported as malicious, Snort IDs (SIDs) indicated for this particular request are analyzed, and a corresponding vector of SIDs is created (this part of the procedure is presented in Section 4.1). All HTTP requests with the identical SID set (i.e., the same vector) are put into the same request group. As presented in Figure 3, SIDs can overlap between the request groups (see, e.g., SID 1); however, only unique SID vectors are treated as distinct. Therefore, {SID 1} and {SID 1, SID 2} groups are treated as different groups, as well as distinct from the {SID 1, SID 2, SID 3} request group. The motivation behind it is that the HTTP requests triggering similar but a bit different set of IDS rules are also a little different from each other.

If not specified otherwise, statistics of the HTTP features are calculated based on the request groups and not based on single requests. An example of statistic calculation is presented in Figure 4. When considering how often request methods are prevalent, it must be verified in how many of the request groups a particular method is present. Every request in a request group is checked; if in all of them the method is present, the request group is treated as one entity from the statistical point of view. In the example presented in Figure 4, such a situation occurs for request groups 1 and 3 with the *GET* method and for the request group 4 with the *POST* method. In the request group 2, both *GET* and *POST* methods are present; thus, such a request group is reported as having multiple values. Depending on the feature, such cases are rather infrequent. Final results for the above-mentioned example indicate that the *GET* method was present in 50% of request groups, *POST* in 25%, and multiple values were present in 25% of request groups.

It is worth noting that the statistics of the benign dataset are calculated directly on requests—in this case, the requests are not grouped as they do not trigger IDS alerts; thus, it is impossible to group them.

Request groups are further divided depending on the type of malware they represent. The main idea behind such a presentation of results is to provide potential insights into characteristics of various malware categories, which can often demonstrate different operational behavior.

As in the example introduced earlier, statistics of the *GET* request occurrences are calculated using request groups of a particular malware category. The algorithm is the same as the one described before; i.e., every request in a request group is checked, and if all of them are *GET* requests, the request group of the category is treated as one unit for the statistics. When in 2 out of the 4 request groups of the category a certain feature is present, the results will show its 50% occurrence in this category.

The name of the related malware was obtained from the IDS alerts and was used for classification purposes. The request groups were divided into 20 categories, presented in Table 2, along with the number of request groups in each category. These categories were based on information provided by the IDS rule comments, information from

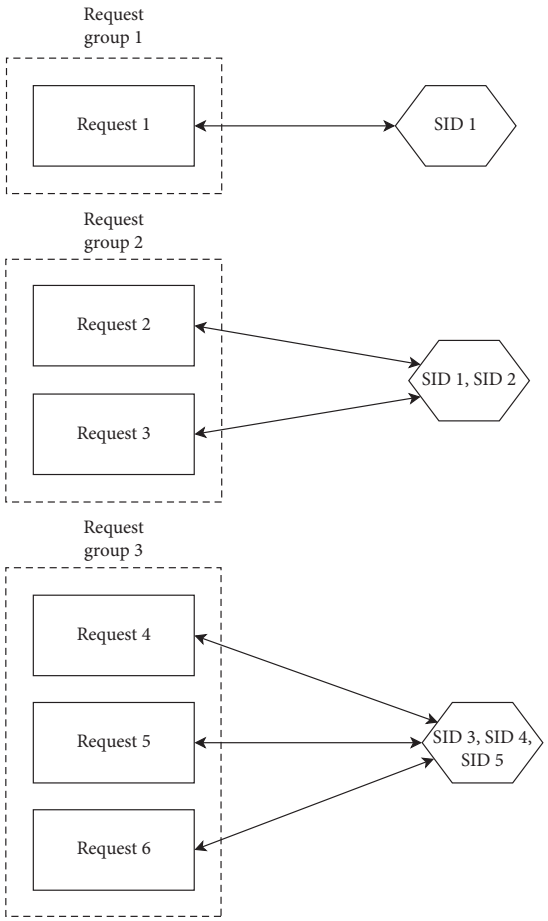


Figure 3: An example of assigning HTTP requests into request groups based on the Snort ID rules reported by this IDS.

malware dissection articles from the Internet, and from own experience. Request groups were labeled semimanually.

Many of the malware categories mentioned in Table 2 are self-explanatory, e.g., *Banker*, *Spambot*, *Ransomware*, or *RAT*. However, some other classes need to be explained in more detail. The *IP check* category groups requests which were sent to the IP address identification services. In that way, malware typically checks the external IP address of the infected machine or whether there is an Internet connection available. The *UA problem* category contains only requests, which were alerted by the IDS as a problematic *User-Agent* header value but without information about the malware family. The *Downloader* type groups all malware families which are used to download other malware. This class is different from *Downloader/JS*, where in the latter one, the actual code is a JavaScript, while in the former one, it is a binary file (EXE file). Similar to these categories is *Maldoc*, where additional malware is downloaded using malicious documents, for example, Microsoft Word or Excel macros. When the IDS labeled a request as a malware download, but no information about the malware family name was

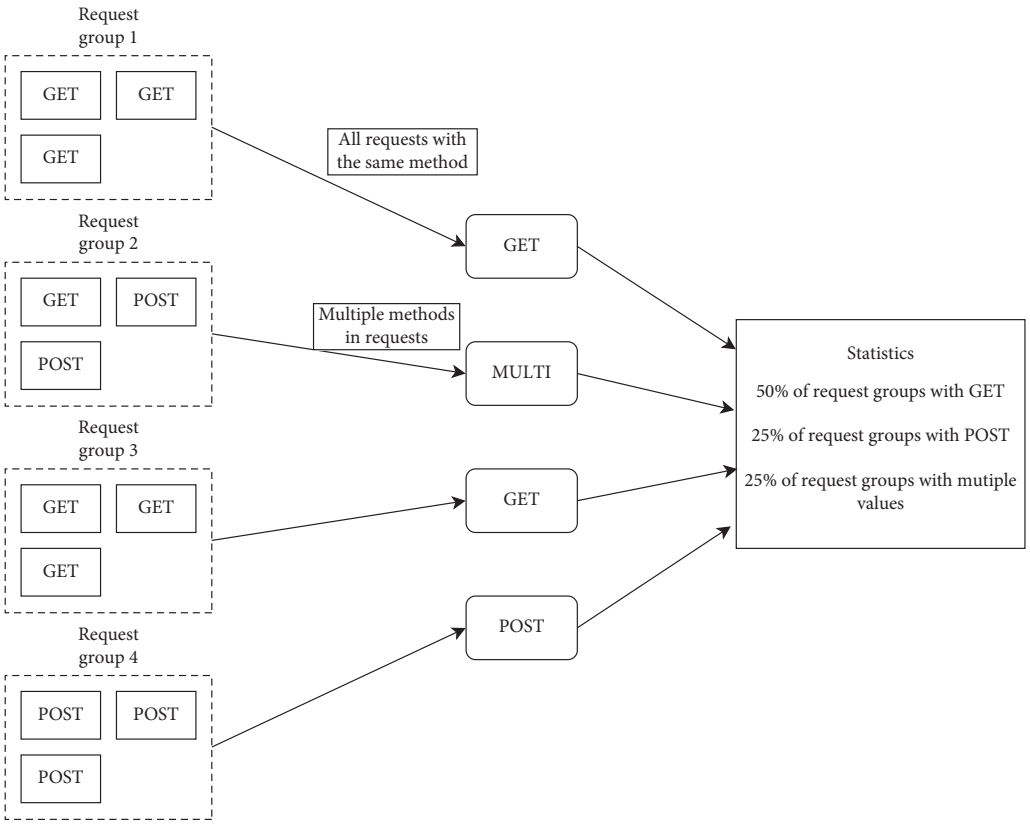


Figure 4: An example presenting statistic calculation methodology used for result analysis.

Table 2: Malware categories used to organize the obtained experimental results.

Category	Description	Number of request groups
Downloader	Downloading other malware	134
Banker	Banking Trojan	125
Trojan	Trojan malware	117
Ransomware	Crypting files and demanding ransom	85
Stealer	Stealing users' information	45
PUA/Adware	Potentially unwanted applications or adware	30
IP check	Checking IP address or connectivity	28
UA problem	Problem with *User-Agent* header value	26
DDoS	DDoS attack malware	24
Spambot	Sending spam e-mails	20
Malicious download	Downloading other malware	20
Miner	Cryptocurrency mining	18
Maldoc	Downloading other malware	16
Clicker	Ad and link clicking	13
Downloader/JS	Downloading other malware	12
Backdoor	Backdoor Trojan	11
RAT	Remote access Trojan	9
Bruteforce	Bruteforcing, e.g., login panels	9
Other	Other malware	8
Keylogger	User key stroke logging	6

provided, the request was treated as the *Malicious download* type. Request groups of Trojan malware which cannot be assigned to any other specialized category (such as Stealer, Banker, or Clicker) were treated as the catch-all *Trojan* kind. Finally, the group request which could not be incorporated into any other category is ascribed to the *Other* class.

The reason behind such a request arrangement (both in request groups and in malware categories) is to limit the effect of inequality of the number of requests between malware families. For example, the *Locky* ransomware family is represented in our dataset by one of the biggest number of requests (ca. 180,000) which constitutes almost 30% of all requests. Without the proposed categorization, such requests would have a tremendous impact on the presented results.

It must also be noted that the quantity bias was not fully eliminated. Even after introduction of request groups, families with a large number of requests can still have a higher number of request groups. The approach taken in this paper regarding malware categories can limit this impact, but it cannot eliminate it completely. The authors of this paper believe that it is a trade-off between bias of the dataset and identification of potential anomalies in the broader datasets.

4.3. Analyzed HTTP Traffic Features. The analyzed HTTP requests' features for the purpose of this paper were assigned into 3 categories, with each of them representing different aspects of the request characteristics: (i) HTTP request structure, (ii) header field values, and (iii) HTTP request payload feature groups. Such an approach was based on the authors' malware behavior analysis experience and previous works of other researchers, as discussed in Section 2. Table 3 outlines the relation between the research source and the corresponding feature category. The description of the proposed feature groups is presented below, along with information, in which features were proposed by the authors of this paper, based on their experience. Such features in this paper are marked with "*".

4.3.1. HTTP Request Structure Features. The analysis of the HTTP request structure involves checking the form of the request, i.e., occurring headers, protocol control information, structure of the fields, and, as an extension, TCP protocol destination port of the request. The features are presented in Table 4.

Repetition of some headers is a known method for HTTP requests' smuggling through network devices such as firewalls and web proxy servers (cf. [33]). In this research, it is utilized to identify errors of malware developers, such as unskillful change in the header value.

4.3.2. Header Field Value Features. Header field values were examined in order to verify whether any of them are invalid or significantly different from others. Also, the presence of some additional or unusual whitespace characters was verified as well as the presence of non-US-ASCII characters (from this point onward, non-US-ASCII and non-ASCII will be used interchangeably). Evaluation of the *User-Agent* header was performed to obtain a list of names which malware presents itself to the server. During analysis of the *Host* header value, the type of the value was determined; it was verified whether it was an IP address, domain name, or some other value. The feature list is presented in Table 5.

The *Host* header value was analyzed for the value types as presented in Table 6.

Host header value is frequently analyzed in the existing works (for example, Li et al. [19]). In this research, it was inspected to establish its value type, including error values. As such, according to the best of the authors' knowledge, it is the first attempt to provide such information in a general manner.

4.3.3. HTTP Request Payload Features. Analysis of the payload data includes features as presented in Table 7.

Its evaluation was performed on the data after decoding/decompression and dechunking and not on the data as seen on the wire. The reason behind it was to analyze the final payload, excluding the influence of compression and chunking mechanisms on the analyzed payload data. Please note that in the "presence of non-ASCII characters in the payload" feature, "non-ASCII" is meant as "non-US-ASCII," and it will be used in the shorter form in this text.

4.4. Data Sources. Two data sources were used in the conducted investigations. As the sources of malicious HTTP traffic, pcap files from CERT Polska's sandbox systems and Malware Capture Facility Project (MCFP) (https://www.stratosphereips.org/datasets-malware) were used. Basic information about these two sources is presented in Table 8.

PCAP files from CERT Polska's sandbox environment were generated by a Windows-based malware, analyzed in 2016–2018. The malware samples originate from automatic systems and incident reports. The former represents systems which collect samples from CERT Polska's internal malware hunting systems and publicly available sources provided by various entities, including Shadowserver (https://www.shadowserver.org/) or Abuse.ch (https://abuse.ch/). The incident reports which provided malware samples were reported mainly by Polish citizens (CERT Polska acts as a Polish national CSIRT) and also by researchers and other entities outside of Poland. All malware samples were acquired during the period of 2016–2018 and represent malware encountered in the wild. The malware analysis system consisted of Windows 7 virtual machines orchestrated by the modified Cuckoo Sandbox system. Main modifications were introduced into hardening the system against anti-VM and anti-analysis techniques and into process monitoring services. MCFP repository is maintained at the Czech Technical University in Prague and consists of pcap files from the long-term Windows malware observations. Both repositories represent popular malware families.

For the legitimate browser traffic, the authors decided to generate it on their own. Various web browsers under control of different versions of the Windows OS were used, as depicted in Table 9. This table also contains a number of analyzed requests for each web browser. The browsers were instrumented using the Selenium automation toolset (https://www.seleniumhq.org/) to visit websites from the list of 500 most popular websites worldwide. The list was created

Table 3: The relation between the research source and the analyzed HTTP request feature groups.

Feature group	Research sources
HTTP request structure	Montero [14], Cuckoo, Calzarossa et al. [13], Rossow et al. [11], Nelson [12], Li et al. [21]
Header field values	Montero [14], Lewis [17], Mizuno et al. [18], Calzarossa et al. [13], Li et al. [19], Kheir [20], Rossow et al. [11], Nelson [12], Li et al. [21], Perdisci et al. [22]
HTTP request payload	Perdisci et al. [22]

Table 4: List of HTTP requests' structure features.

Feature name
HTTP protocol version
Request method
Repetitions of the header (two header fields with the same name)*
Lack of colon in the header field*
Number of headers in the request
Frequency of the headers' occurrence
Misspellings of the header names
Presence of request pipelining*
TCP destination port in the request

Features proposed by the authors of this paper are marked with * (an asterisk).

Table 5: List of header field value features.

Feature name
First character of the header field is a whitespace
Whitespace before CRLF tag
Space before colon, semicolon, or comma
New line character other than CRLF*
Double space
Nonstandard whitespace characters in the header field*
Non-ASCII value in the header*
Accept-Language header value
Accept-Encoding header value
Connection header value
Host header value*
User-Agent header value

Features proposed by the authors of this paper are marked with * (an asterisk).

Table 6: List of *Host* header value types.

Host header value type
IP address
Domain name
IP address with the port number
Domain name with the port number
Error in the domain
Other value

using the Alexa top 1 million websites worldwide (http://s3.amazonaws.com/alexa-static/top-1m.csv.zip). The websites were accessed between 9 and 15 February 2017 and between 13 and 18 October 2017, depending on the browser.

Table 10 presents the top 5 malware families in the categories grouped by the number of request groups. It should be noted that 18.67% of request groups were sent by an unknown malware. In the table, such request groups are marked as *No-name*.

Table 7: List of HTTP request payload features.

Feature name
Payload data length
Payload entropy value
Presence of non-ASCII characters in the payload*
Presence of non-POST requests with the payload*
Presence of *Referer* header in the POST request

Features proposed by the authors of this paper are marked with * (an asterisk).

5. Experimental Results

In this section, experimental results of analysis of features presented before in Section 4.3 are outlined. The presentation of the results uses the categorization introduced there.

5.1. HTTP Request Structure Features. Some of the analyzed features did not show any results in the malware and browser HTTP traffic. These are the repetitions of the header (two header fields with the same name) and presence of pipelined requests; i.e., multiple requests are sent without waiting for their corresponding responses. The lack of colon in the header field was not observed in malware traffic and was present in only 4 requests of Internet Explorer 11 on Windows 7 and Windows 8.1 (two in each browser), i.e., in about 0.01% of requests. Also, analysis of HTTP request methods showed that it cannot be directly used to distinguish between malware and browser traffic. It is however indicated that browsers mostly sent the *GET* request, while a significant portion of malware requests are *POST* requests.

5.1.1. HTTP Version. The results of the analysis of the HTTP protocol version in malware traffic are presented in Figure 5. The majority of the analyzed malware families grouped in categories usually used version 1.0 of HTTP. The highest level of occurrence of the version 1.1 (i.e., 42.22% of requests groups) has the malware in the *Stealer* category, while the other categories have a lower number of request groups for this version of the protocol. Banker, Downloader, and Trojan have about 15% of request groups with version 1.1. RAT and PUA/Adware about 10%. There are 6 categories with nonzero levels up to 10% of requests groups. Finally, 6 categories of malware do not have any requests with version 1.1.

Table 8: Basic information about malicious pcap repositories.

Feature	CERT.pl	MCFP	Sum
No. of pcaps in repository	36,268	117	36,385
No. of pcaps with HTTP network traffic	26,042	91	26,133
No. of pcaps with HTTP network traffic containing requests alerted by IDS	22,630	67	22,697
No. of reported IDS alerts	2,133,682	425,441	2,559,123
No. of reported IDS alerts assigned to requests	405,116	238,805	643,921
No. of unique alerted IDS rules	578	139	642

Table 9: Networking environments in which HTTP traffic was analyzed.

Browser name	Operating system	Abbreviation	Number of requests
Microsoft Edge	Windows 10	Edge Win10	17,912
Google Chrome	Windows 7	Chrome Win7	30,621
Mozilla Firefox (Adobe Flash Player installed)	Windows 7	Firefox-FP Win7	18,705
Mozilla Firefox	Windows 7	Firefox Win7	28,178
Microsoft Internet Explorer 11	Windows 7	IE11 Win7	30,799
Google Chrome	Windows 8.1	Chrome Win8.1	23,967
Mozilla Firefox	Windows 8.1	Firefox Win8.1	18,153
Microsoft Internet Explorer 11	Windows 8.1	IE11 Win8.1	20,248

Note. Abbreviations introduced here are used in the paper to refer to the specific environments.

The same analysis related to the benign browser traffic showed that only in Internet Explorer-based traffic running under Windows 7 and 8.1 OSs, HTTP requests with version 1.0 occurred (0.01% and 0.08%, respectively). The comparison of the results for malicious and benign traffic shows significant difference in protocol version usage. Therefore, in the authors' opinion, this feature is a good candidate in selection of features distinguishing malware from browsers.

5.1.2. Number of Header Fields. Analysis of the number of headers in the request groups shows that the number of headers varies between 1 and 11. The results are presented in a graphical form in Figure 6. Their analysis shows that in most categories, there were less than 8 headers and up to 6 headers in categories such as *Clicker*, *DDoS*, *Maldoc*, *Miner*, *PUA/Adware*, or *Spambot*. In many categories, 5 headers in a request is a dominant value.

Most categories have request groups with multiple values of headers' number. These include 4 categories with more than 40% of request groups (*Bruteforce, IP check, Keylogger,* and *Malicious download*). All of them were analyzed further, and results indicate similar header number ranges as in the single header number value request groups.

The number of headers in browser requests is in the range from 0 to 24 headers. However, for every browser, the number of headers was in the range between 0 and 11 in at least about 99% of requests. Results for this range are presented in Figure 7 where percentage results of the number of headers in the browser traffic requests are illustrated. The most common number of headers is 7 and 8.

However, the ranges of the number of headers for malware and browser traffic overlap, and their distributions are different. As already mentioned, in the benign traffic, most of the values are close to two maxima (7 or 8 headers in a request). For malicious traffic, the majority of requests has up to 6 headers. From this perspective, the number of headers in the request can be perceived as a useful feature to distinguish malware and browser HTTP traffic.

5.1.3. Header Occurrence. The top 10 headers sorted by their average frequency of occurrences in the benign traffic are presented in Table 11. The first 7 headers occurred in all browsers in at least about 90% of the requests. Some of the well-known headers did not occur in the top 10, for example, *Origin* (3.97% of all requests in all browsers), *Content-Type* (1.21% of all requests), *Cache-Control* (1.18% of all requests), or *Content-Length* (0.95% of all requests). Nonstandard headers, which begin with prefix *X*, were also observed. Some of them are relatively known, e.g., *x-flash-version*, but others are server platform specific, e.g., *X-TeaLeaf-Browser-Res*. One of the headers observed in the benign dataset was particularly interesting. The "_" (an underscore) header was present only in two requests sent to *unid.go.com* on Windows 7 OS by Google Chrome and Microsoft Internet Explorer 11. This network traffic is associated with the content delivery networks (CDN) operations. Additionally, some of the well-known headers were observed written in varying cases, for example, *Authorization* and *authorization*, *Content-Type*, *Content-type*, and *content-type*, and *Accept* and *accept*. Generally, lower case versions were less frequent.

Table 12 summarizes the presence of particular headers in the requests of malware traffic. The values present the top 10 headers regarding the percentage of all malware categories where the header appeared. Percentages were counted in the request groups in which the header was present in all requests.

Analysis of unique header names found in malicious traffic shows that besides well-known headers, their versions written in lowercase were also present, for example, *accept-Language* or *Content-type*. Some of the header names cannot

Table 10: Top 5 malware families in categories grouped by the number of request groups.

Family name	Number of request groups
Backdoor	
Htbot	3
GrayBird	2
Dimnie	2
Zeprox	1
Votwup.D	1
Mokes	1
Banker	
Ursnif	27
Dreambot	24
Chthonic	12
Emotet	11
Kronos	10
Bruteforce	
No-name	6
Pifagor	2
Clicker	
KOVTER	6
Zeroaccess	4
Sefnit	2
Miuref/Boaxxe	1
DDoS	
DirtJumper	17
MegalodonHTTP	4
Madness	2
MedusaHTTP	1
Downloader	
Pony	21
Nemucod	19
SmokeLoader	17
Locky	12
Zbot	11
Downloader/JS	
No-name	8
Cryxos	4
IP check	
No-name	28
Keylogger	
AgentTesla	3
Keybase	2
KeyLogger.acqh	1
Maldoc	
No-name	16
Malicious download	
No-name	20
Miner	
No-name	11
Adylkuzz	4
1ms0rry	2
Smominru	1
Other	
FakeAlert.jh	3
Ratankba	1
Psiphon	1
No-name	1
DustySky	1
PUA/Adware	
Wizzcaster	3
InstallCapital	3
BubbleDock	3
Sureseeker	2
OfferCast	2
Ransomware	
Locky	38
AlphaCrypt	8
PadCrypt	4
Sage	3
Fatboy	3
RAT	
Quasar	2
XPCSpyPro	1
TViewer	1
Teamspy	1
ShinoBot	1
Spambot	
Kelihos.F	8
Necurs	5
XnxxAgent	3
Sality	3
Tofsee	1
Stealer	
AZORult	11
Loki	10
FormBook	6
WernikStealer	2
Hawkeye	2
Trojan	
Zbot	29
No-name	16
Andromeda	12
Graftor	7
Betabot	6
UA problem	
No-name	26

be found in any official documentation, for example, *Filename*, *Idle-time*, *Content-Key*, or *Server-Key*. One header *user-* looks like it was created to mimic the *User-Agent* header, but for some reason it remained unfinished.

It must also be noted that no misspellings of header names were found in the observed malicious and benign traffic. The *user-* cannot be categorized as misspelling, but in some way it proves the observation that sometimes malware developers do make errors.

In both HTTP traffic datasets, the header names with alternative case spellings were observed, for example, *Content-Type* and *Content-type*. RFC 7230 does not prohibit such a usage, stating that header names are case insensitive. However, the observed traffic demonstrates that upper-cased first characters are more popular, regardless of the type of the traffic dataset (benign/malicious). The occurrence of the lower-cased version of the header names is low in both malware and

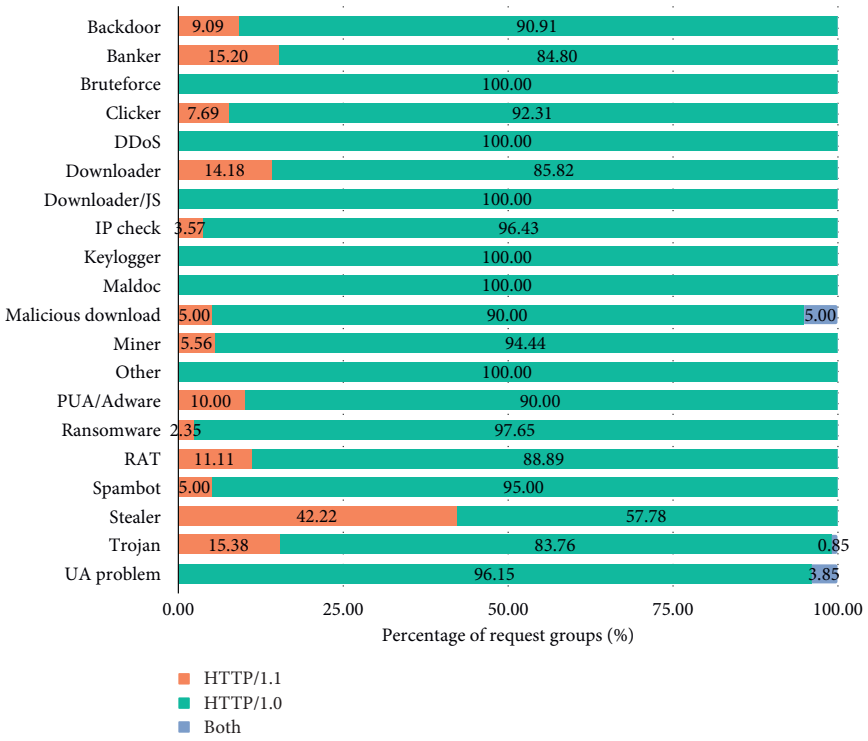


Figure 5: HTTP protocol version (malicious traffic).

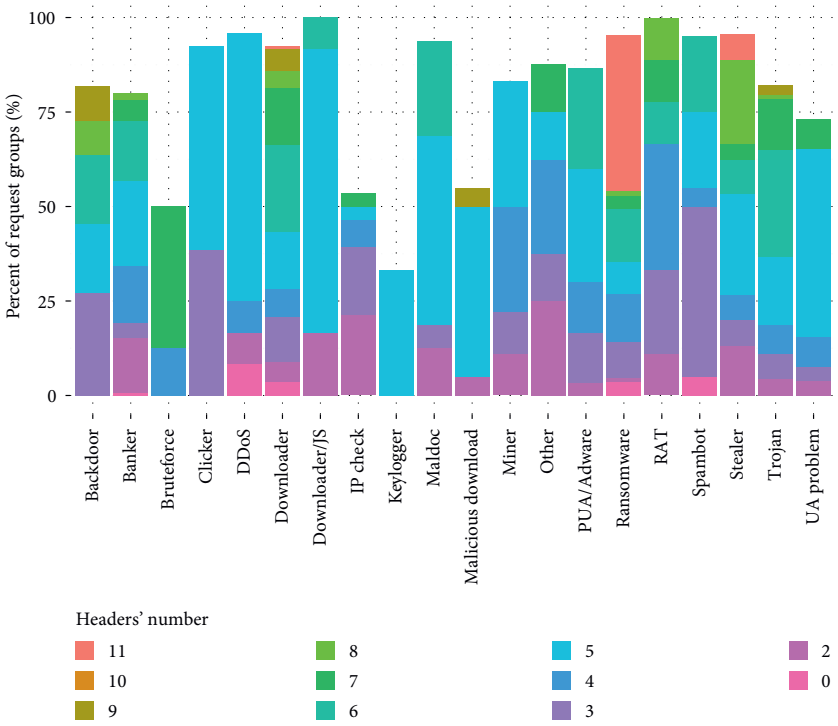


Figure 6: The number of headers in a request (malicious traffic).

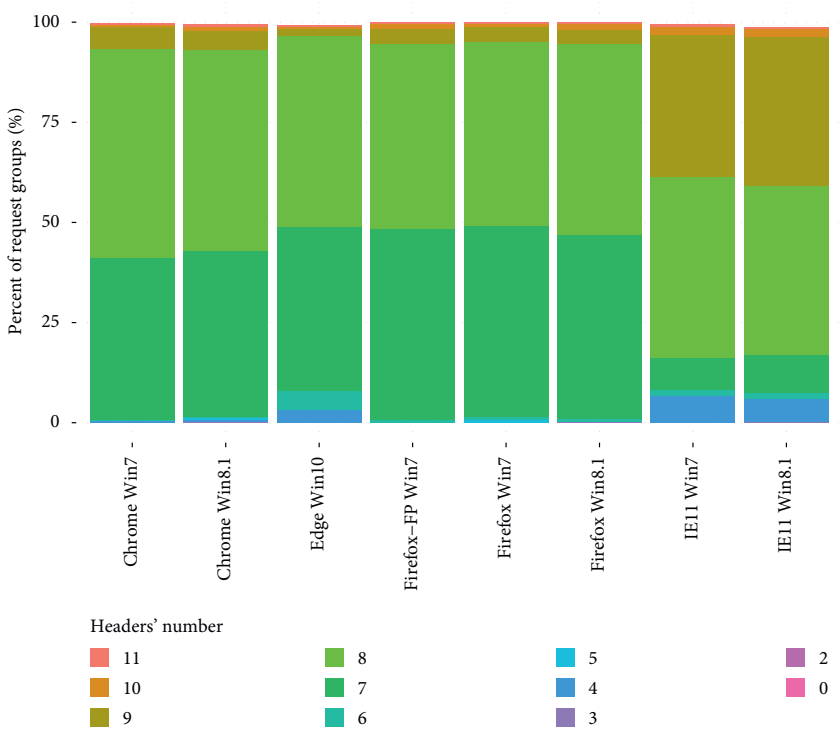


Figure 7: The number of headers in a request (benign traffic).

Table 11: Top 10 headers in a request (benign traffic).

Browser	Header name-percentage of requests									
	"Host"	"User-Agent"	"Connection"	"Accept"	"Accept-Encoding"	"Accept-Language"	"Referer"	"Cookie"	"DNT"	"Upgrade-Insecure-Requests"
Edge Win10	100.00	99.89	100.00	99.86	96.80	95.46	91.05	48.64	1.07	0.00
Chrome Win7	100.00	100.00	100.00	99.79	99.62	99.16	95.62	55.19	0.00	6.12
Firefox-FP Win7	100.00	100.00	100.00	100.00	99.96	99.84	93.97	48.30	0.00	7.35
Firefox Win7	100.00	100.00	100.00	99.99	99.92	98.12	93.86	48.05	0.00	5.64
IE11 Win7	99.99	99.99	99.99	99.98	92.90	91.98	88.33	43.88	78.48	0.00
Chrome Win8.1	100.00	100.00	99.78	99.46	99.39	97.78	93.64	52.02	0.00	6.81
Firefox Win8.1	100.00	100.00	99.82	99.83	99.78	99.77	94.01	49.62	0.00	7.31
IE11 Win8.1	99.99	99.99	99.74	99.70	93.91	92.98	88.65	44.02	80.26	0.00
Average	100.00	99.99	99.93	99.83	97.71	96.77	92.41	48.83	21.54	4.13

browser traffic and therefore is not distinctive enough to show the general difference between the malicious and benign traffic. Nevertheless, it can be more useful for distinction from the perspective of individual malware families.

Based on the presented analysis, it can be concluded that the presence of some particular headers can be used as a feature for distinction between malicious and benign traffic. The list of such headers should include those indicated as the most popular ones in the browser traffic: *Connection*, *Accept*, *Accept-Encoding*, *Accept-Language*, and *Referer*. These headers appear in the analyzed malware traffic less frequently.

Some previous works have been already performed when it comes to the usage of the header order in malware

Table 12: Top 10 headers present in requests (malicious traffic) sorted by % of all categories where they appeared.

Category	Percentage of requests in category									
	"Host"	"User-Agent"	"Connection"	"Accept"	"Accept-Encoding"	"Cache-Control"	"Content-Length"	"Content-Type"	"Accept-Language"	"Cookie"
Backdoor	100.00	100.00	45.45	45.45	27.27	54.55	45.45	54.55	27.27	9.09
Banker	100.00	73.60	75.20	13.60	1.60	77.60	60.80	19.20	4.80	4.80
Bruteforce	100.00	100.00	87.50	100.00	12.50	0.00	87.50	87.50	0.00	25.00
Clicker	100.00	100.00	30.77	7.69	7.69	46.15	61.54	46.15	0.00	0.00
DDoS	100.00	83.33	87.50	0.00	0.00	8.33	4.17	4.17	0.00	0.00
Downloader	100.00	90.30	75.37	55.22	32.84	44.03	36.57	30.60	20.15	0.75
Downloader/JS	100.00	83.33	100.00	83.33	83.33	0.00	0.00	0.00	8.33	0.00
IP check	100.00	67.86	75.00	32.14	28.57	14.29	0.00	0.00	28.57	7.14
Keylogger	100.00	66.67	0.00	0.00	0.00	16.67	66.67	66.67	0.00	0.00
Maldoc	100.00	100.00	81.25	81.25	81.25	6.25	0.00	0.00	25.00	0.00
Malicious download	100.00	75.00	80.00	60.00	40.00	35.00	10.00	10.00	5.00	0.00
Miner	100.00	77.78	50.00	11.11	27.78	0.00	38.89	38.89	0.00	0.00
Other	100.00	75.00	62.50	12.50	12.50	12.50	12.50	25.00	12.50	12.50
PUA/Adware	100.00	80.00	66.67	23.33	13.33	43.33	40.00	33.33	0.00	3.33
Ransomware	100.00	80.00	71.76	62.35	47.06	70.59	77.65	71.76	42.35	1.18
RAT	100.00	88.89	55.56	22.22	22.22	33.33	44.44	44.44	0.00	11.11
Spambot	100.00	70.00	45.00	5.00	5.00	40.00	75.00	35.00	5.00	0.00
Stealer	100.00	57.78	86.67	35.56	13.33	11.11	66.67	68.89	26.67	0.00
Trojan	100.00	90.60	74.36	47.86	12.82	53.85	52.14	35.90	9.40	2.56
UA problem	96.15	92.31	80.77	11.54	15.38	19.23	11.54	11.54	0.00	7.69

Note. The header was present in all requests of a particular request group in the malware category.

detection or browser fingerprinting, for example, in the p0f tool (http://lcamtuf.coredump.cx/p0f3/). The idea behind it was to check the order in which the headers occur in the HTTP request and to identify the application which sends it. The authors of this paper believe that this problem has not been fully analyzed, and more research is needed.

5.1.4. Destination Port. Destination ports of requests in malicious traffic were also investigated. Unsurprisingly, most of the requests were sent on port 80. 7 malware categories sent requests to other ports, but even in these situations, at least 88% of the request groups used port 80. Two categories (*Banker* and *Ransomware*) sent requests on port 443, which is a registered port for the HTTPS protocol. This behavior can be seen as anomalous regardless of the application type which sent such requests.

Destination ports of requests in benign traffic were also analyzed. It occurred that every browser sent over 99.8% of requests to port 80. However, some other ports were also discovered, e.g., 443, 8080, 880, and 8050.

When comparing traffic results for malware and browsers, one can see that both categories send requests mainly on port 80. Other destination ports occur, but they are not so frequent. The main difference between the browser and the malware HTTP traffic is that the malware uses ports of higher numbers, for example, higher than 10,000 in the *Downloader* category and higher than 40,000 in the *Ransomware* category. This difference can be seen only for single-request groups, and it cannot be confirmed as regular. Thus, the analysis of the utilized destination ports cannot be conclusive to distinguish between the malicious and benign traffic.

Nevertheless, usage of some ports for HTTP traffic is improper, for example, port 443 which is registered by IANA for the HTTPS protocol. Such situation can be anomalous on its own, regardless of the type of network traffic, and is usually alerted by the network monitoring systems.

5.2. Header Field Value Features. In this section, different features of the header field values were investigated. The conducted experiments revealed that some of the features initially selected to inspect (see Section 4) were not present at all in the analyzed HTTP requests. This includes the following features: (i) the presence of the space at the beginning of the header field, (ii) the occurrence of space before colon, and (iii) the appearance of the space before semicolon. Considering the above, the obtained results for these 3 features are omitted. Some other features were observed in the analyzed traffic. They did not however give any significant results that can be utilized for distinguishing between malware and browser traffic because they were rare. Whitespace character before CRLF tag was not present in the browser traffic, but it was present in network traffic of 5 malware categories (of which 2 categories were exceeding 10% of request groups). The next feature is the presence of a space character appearing before a comma in the header field. Its analysis revealed that it was absent in the browser traffic and present only in about 2.35% of request groups of the *Ransomware* category. The new line character other than CRLF was not present in malware traffic, but it was observed

in the browser traffic of Internet Explorer 11 on Windows 7 and 8.1 in 2 and 13 requests, respectively. Both values are below 0.1% of all requests for both browsers. The next feature which gave limited results is the presence of a double space in the header field. It was not observed in the malicious traffic; with regards to the browser traffic, only Google Chrome (both on Windows 7 and 8.1) did not send requests with double space in the header field. Mozilla Firefox on Windows 7 presented the highest percentage of such requests (0.21%), but overall, the percentages are low. Finally, nonstandard whitespace characters were not observed in browser requests, and in malicious traffic, it was present only in 1.18% of *Ransomware* request groups. Additionally, analysis of values of *Accept-Language*, *Accept-Encoding*, and *Connection* headers did not indicate any distinguishing features between browser and malware traffic. Thus, the numerical results will be omitted for brevity of the text.

5.2.1. Non-ASCII Characters in the Header Field. Furthermore, the presence of the non-ASCII characters in header fields in the malicious traffic was analyzed. This feature was observed in two malware categories: *Backdoor* (3.57% of request groups) and *Ransomware* (1.18% of request groups). Additionally, in 3 request groups of the *Ransomware* category (additional 3.53%), the feature was present irregularly.

Non-ASCII character in the header field in benign HTTP traffic was observed only for 3 browsers: Chrome, Firefox, and Internet Explorer running on Windows 7 OS. However, it was only one request per browser, which is less than 0.01% of all requests in the respective sets. It was caused by the presence of Polish characters.

It must also be noted that the presence of the non-ASCII character in the header field can be treated as an anomaly itself. It however occurs sporadically in both malicious and benign traffic. It can be considered as an indicator of anomalous traffic, but in the presented form, it cannot be used as a general rule to distinguish malware and browser HTTP traffic.

5.2.2. Host Header. The obtained *Host* header values in the malicious traffic are presented in Table 13. The domain is present as the main value in most of the malware categories. However, for some categories, also the IP address value is noticeable. These include *Ransomware*, *Spambot*, *Clicker*, and *Miner* categories. Some categories also have request groups with multitype values, for example, *Maldoc* has the highest percentage among all categories, i.e., 31.25%.

The analysis of actual values in the requests with value types defined as *Error in domain* and *Other* was also performed. The results indicate that the domain names contained suffixes such as *.bit*, *.xn–p1ai*, additional "." characters (*.com.*), or were malformed, for example, *7M5 us* or *5t9AR us*. Also, the malformed IP address and the port pair were identified (*5.141.22.43:13404*).

The results of the feature analysis for the benign browser traffic show that in the majority of requests, the domain is present in the *Host* header value, regardless of the browser and OS used. In all browsers, such a value was present in at least 99.8% of requests. Other value types include IP address (maximum value of 0.07% for Chrome browser on Windows 7), domain and port (maximum value of 0.1% in case of Firefox browser on Windows 8.1), and IP address and port. However, the latter ones are negligible for all browsers.

The comparison of results of value types in the *Host* header shows that values other than the domain name are more frequently spotted in the malware traffic. This means that this feature can be used in some cases to discern malware and browser traffic. However, the feature is strongly related with the infrastructure used by cybercriminals. Intuition and malware analysis experience suggest that attackers do use some nonstandard addresses for C&C servers. The results show that it does not happen as often as it could be expected. This research does not analyze the purpose of sending particular requests; nevertheless, some of them are sent by malware to benign addresses, for example, as a connectivity check. This could impact the obtained results.

5.2.3. User-Agent Header. The analysis of the *User-Agent* header strings was performed for both traffic types. Some typical values observed in the browser traffic are presented in Table 14.

Table 15 presents the distribution of the *User-Agent* value types in the browser traffic. The values were analyzed based on their similarity to standard values (presented in Table 14) or lack of them in the *User-Agent* header. All browsers used standard values in at least 91% of the requests. Internet Explorer on Windows 7 and Windows 8.1 OSs in about 1% of the requests used values similar to the standard ones but slightly expanded in some fields, e.g., *Mozilla/4.0* (*compatible; MSIE 7.0; Windows NT 6.1; Trident/7.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C*).

Also, these two browsers experienced a higher percentage of values not similar to the standard User-Agent strings (7.35% and 6.37%, respectively). Some significant results were also noted for the Microsoft Edge, Mozilla Firefox on Windows 7, and Google Chrome on Windows 8.1. For all Microsoft browsers (Internet Explorer 11 on Windows 7 and 8.1 and Edge) the main part of the requests with nonstandard values consists of the request sent by modules responsible for downloading certificate revocation lists [34]—*Microsoft-CryptoAPI*. This mechanism is utilized by browsers to download the current sets of revocated X.509 certificates used in the HTTPS protocol communication. Other *User-Agent* header values were also present, for example, *Microsoft BITS* or *Microsoft-WNS* which can be used by OS mechanisms like Windows Update or Windows Push Notification Services from Windows 8 onward. The usage of the *Microsoft BITS User-Agent* found in the Chrome browser HTTP traffic (both for Windows 7 and 8.1) can be attributed to the update mechanism of this browser.

Four browsers requests without the *User-Agent* string were present. In overall, this was applied to less than 0.1%. These requests were sent by system mechanisms or generated along with the web page activity.

Table 13: *Host* header values (malicious traffic).

Category	Domain % of all	IP % of all	IP + port % of all	Domain + port % of all	Error in domain % of all	Other % of all	Multi % of all
Backdoor	72.73	9.09	0.00	0.00	0.00	0.00	18.18
Banker	70.40	9.60	8.00	0.00	6.40	0.00	5.60
Bruteforce	87.50	0.00	0.00	0.00	0.00	0.00	12.50
Clicker	69.23	23.08	0.00	0.00	0.00	0.00	7.69
DDoS	100.00	0.00	0.00	0.00	0.00	0.00	0.00
Downloader	76.87	6.72	2.24	0.75	2.24	0.75	10.45
Downloader/JS	100.00	0.00	0.00	0.00	0.00	0.00	0.00
IP check	100.00	0.00	0.00	0.00	0.00	0.00	0.00
Keylogger	66.67	16.67	0.00	0.00	0.00	0.00	16.67
Maldoc	68.75	0.00	0.00	0.00	0.00	0.00	31.25
Malicious download	80.00	10.00	0.00	0.00	0.00	0.00	10.00
Miner	72.22	22.22	0.00	5.56	0.00	0.00	0.00
Other	75.00	0.00	25.00	0.00	0.00	0.00	0.00
PUA/Adware	100.00	0.00	0.00	0.00	0.00	0.00	0.00
Ransomware	40.00	52.94	1.18	0.00	0.00	0.00	5.88
RAT	77.78	11.11	0.00	0.00	11.11	0.00	0.00
Spambot	40.00	60.00	0.00	0.00	0.00	0.00	0.00
Stealer	68.89	13.33	4.44	0.00	0.00	0.00	13.33
Trojan	65.81	17.09	3.42	0.85	0.00	0.00	12.82
UA problem	80.77	11.54	0.00	0.00	0.00	0.00	7.69

Table 14: The standard values of the *User-Agent* header (benign traffic).

Browser	User-Agent value
Edge Win10	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.2743.116, Safari/537.36 Edge/15.15063
Chrome Win7	Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36
Firefox-FP Win7	Mozilla/5.0 (Windows NT 6.1; rv:51.0) Gecko/20100101 Firefox/51.0
Firefox Win7	Mozilla/5.0 (Windows NT 6.1; rv:51.0) Gecko/20100101 Firefox/51.0
IE11 Win7	Mozilla/5.0 (Windows NT 6.1; Trident/7.0; rv:11.0) like Gecko
Chrome Win8.1	Mozilla/5.0 (Windows NT 6.3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36
Firefox Win8.1	Mozilla/5.0 (Windows NT 6.3; rv:56.0) Gecko/20100101 Firefox/56.0
IE11 Win8.1	Mozilla/5.0 (Windows NT 6.3; Trident/7.0; rv:11.0) like Gecko

Table 15: The distribution of the *User-Agent* header value types (benign traffic).

Browser	Main UA % of all	Similar to main UA % of all	No UA % of all	Other % of all
Edge Win10	96.81	0.00	0.11	3.08
Chrome Win7	99.19	0.00	0.00	0.81
Firefox-FP Win7	99.84	0.00	0.00	0.16
Firefox Win7	98.12	0.00	0.00	1.88
IE11 Win7	91.53	1.11	0.01	7.35
Chrome Win8.1	97.91	0.00	0.00	2.09
Firefox Win8.1	99.77	0.00	0.00	0.23
IE11 Win8.1	91.99	1.63	0.01	6.37

In the end, the authors decided to leave such requests in the dataset as it is not known with certainty which requests were sent by the browsers and which were not. The value of the *User-Agent* header could be misleading, or assumption about the system service could be erroneous. Additionally, the number of nonbrowser requests is not high and thus should not introduce much noise into the results.

For the *User-Agent* header, the analysis performed on the malicious HTTP traffic uncovered 6218 unique values of the *User-Agent* header. These values were further analyzed in order to establish well-known browser results as malware developers typically try to mimic the behavior of the benign software. The results of this analysis are presented in Figure 8, and they are grouped by the popular browser names and lack of the *User-Agent* header or the nonstandard value. If the requests carry many different values from any of these classification groups, they are classified into the *Misc* group.

From the 4 browsers indicated in Figure 8, most of the requests include Internet Explorer or the Firefox *User-Agent* string. Also, only 4 categories (*Backdoor, Bruteforce, Downloader/JS,* and *Spambot*) do not have requests without the *User-Agent* header. Some categories have a high percentage of the *User-Agent* values other than those 4 standard ones, e.g., *Miner, Other, PUA/Adware, RAT*, or *Stealer*.

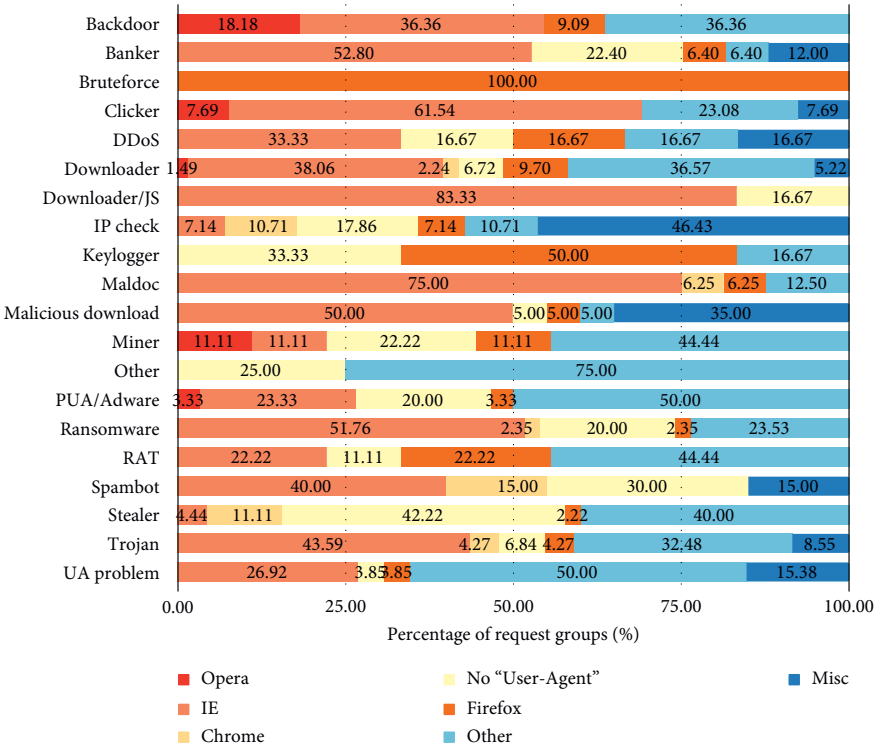


Figure 8: Browser string as reported in the *User-Agent* header (malicious traffic).

Nonstandard *User-Agent* header values discovered in the malicious HTTP traffic present values of the software modules/libraries, such as *LuaSocket 2.0.2*, *AutoIt*, or *python-requests/2.12.4*. Some others include meaningless values, e.g., *W1pbbA((*, *EMSFRTCBVD*. Others upload some system information, presented below with regular expression or obfuscated for privacy purposes: *regexp: ^{[A-Z0-9]{8}-[A-Z0-9]{4}-[A-Z0-9]{4}-[A-Z0-9]{4}-[A-Z0-9]{10}}$*, *C:Users[user's name]AppDataRoaming v2o5g0Ie5itemp.zip*. Finally, some of them present directly their name and purpose, e.g., *TrickLoader* or *Botnet by Danij*.

It must also be noted that additional examination should be applied to the requests without the *User-Agent* header. They are hard to be found in the browser HTTP traffic, but they are present in the majority of malware traffic—from 3.85% of the request groups of the *UA problem* category up to 42.22% for the *Stealer* category. The authors believe that the lack of the *User-Agent* header can be used to distinguish between the malicious and benign HTTP traffic.

5.3. HTTP Request Payload Features. The analysis of the payload data length did not give any significant results in distinction between malware and benign traffic. However, the authors of this paper believe it should be analyzed with a more specialized approach than used in the study, giving more specific results.

5.3.1. Payload Entropy. In Table 16, the results of analysis of the request payload entropy for malicious traffic are presented. As was the case with the payload length, the statistics are based on the values of the payload entropy inside malware categories, but the values are not organized into request groups. On the other hand, the investigation of the same feature in the browser traffic is presented in Table 17.

The comparison of the obtained results demonstrates that many malware families achieved higher levels of the payload entropy. When comparing the median, mean, quartile, and maximum values, 11 out of 15 malware categories have higher mean and median values of the payload entropy than in the benign HTTP requests. The maximum value of the browser traffic payload entropy is 6.13 bits (in Chrome browser on Windows 8.1), whereas in the malicious traffic in 4 categories (i.e., *Downloader, PUA/Adware, Spambot*, and *Trojan*), the value achieved almost the highest possible value of 8 bits. The minimum values of entropy of 1.0 bits in both malware and browser traffic are caused by the requests with a very small payload size (1-2 bytes).

Figure 9 allows visual comparison of malware categories and browser traffic, and it presents the boxplot diagram of the corresponding payload entropy. Categories such as *Bruteforce, DDoS, Keylogger, RAT*, and *UA problem* almost overlap with the benign dataset entropy range when analyzing their interquartile range. For *Ransomware* and *PUA/Adware* categories, the median value is slightly smaller than

Table 16: The payload entropy statistics for HTTP requests (malicious traffic) in bits.

Category	Median	Mean	1st quartile	3rd quartile	Min value	Max value
Backdoor	5.86	5.85	5.86	5.86	4.08	5.96
Banker	5.42	6.11	5.42	7.35	1.00	7.86
Bruteforce	4.27	4.32	4.27	4.35	4.24	7.63
Clicker	5.91	5.88	5.89	5.92	4.29	5.96
DDoS	4.31	4.31	4.31	4.31	4.31	4.31
Downloader	6.15	6.41	5.06	7.76	3.88	8.00
Keylogger	4.96	5.05	4.96	5.14	1.82	7.63
Miner	5.52	5.47	5.51	5.53	3.60	5.99
PUA/Adware	4.21	4.45	4.21	4.21	4.21	7.95
Ransomware	4.44	4.48	4.39	4.48	3.51	7.53
RAT	4.99	4.97	4.78	5.00	4.45	5.89
Spambot	7.10	6.84	6.77	7.16	3.85	8.00
Stealer	6.00	5.24	4.29	6.03	4.11	6.68
Trojan	5.81	5.39	4.36	5.82	1.00	7.99
UA problem	4.98	5.04	4.87	5.13	4.65	5.63

Note. The statistics were counted using all requests in the particular malware category, without being organized into request groups.

Table 17: The payload entropy statistics for HTTP requests (benign traffic) in bits.

Browser	Median	Mean	1st quartile	3rd quartile	Min value	Max value
Edge Win10	4.84	4.74	4.62	4.99	2.73	5.72
Chrome Win7	4.81	4.71	4.42	5.10	1.00	5.93
Firefox-FP Win7	4.72	4.30	3.24	4.92	1.00	5.76
Firefox Win7	4.82	4.72	4.43	5.16	1.00	5.75
IE11 Win7	4.80	4.72	4.43	5.09	2.50	5.94
Chrome Win8.1	4.74	4.39	3.24	4.96	1.00	6.13
Firefox Win8.1	4.83	4.70	4.53	5.02	1.00	5.78
IE11 Win8.1	4.82	4.76	4.59	4.98	2.50	5.72

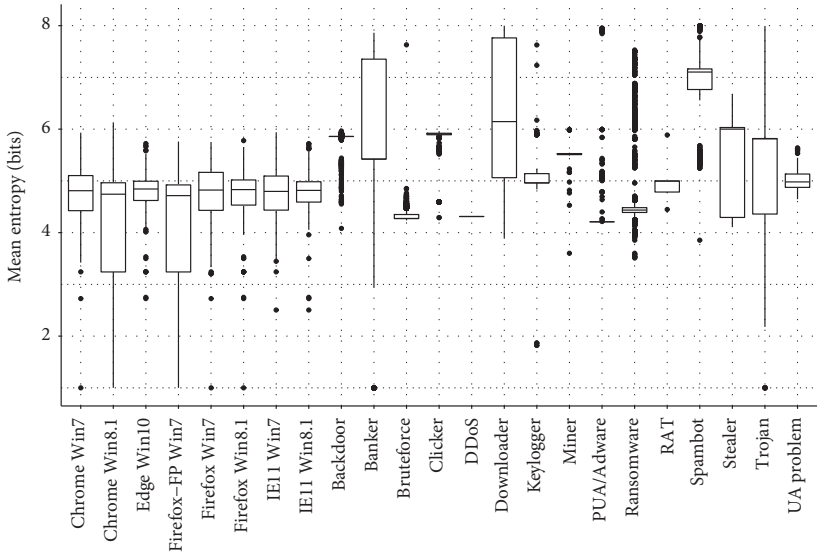


Figure 9: Boxplot diagrams of the payload entropy of malicious and benign HTTP traffic.

for the benign traffic. Both categories however have many outliers. *Backdoor, Clicker*, and *Miner* categories have median values higher than those for the browser traffic. They also have outlying values in the interquartile range of benign traffic. *Stealer* and *Trojan* categories have higher median values than browser HTTP traffic, but still, the interquartile

range partially overlaps with those of browser traffic. Categories such as *Banker, Downloader*, and *Spambot* visually achieve different distribution of values than other categories, and their median values are higher than those of browser traffic.

Overall, the obtained results prove that for many malware categories, the payload entropy is higher than for the browser HTTP traffic. From this perspective, the value of the payload entropy can be used as a feature to differentiate between the malicious and benign traffic. It should also be noted that the above conclusions are made in a general manner and that some particular malware families can exhibit different behaviors.

5.3.2. Non-ASCII Characters in Payload. Figure 10 shows the presence of non-ASCII characters in the payload of malicious HTTP traffic. For the *Banker, Downloader*, and *Spambot* categories, more than 50% of request groups contained non-ASCII characters in the payload data. Additionally, non-ASCII characters were present also in the *Bruteforce, Keylogger, PUA/Adware, Ransomware*, and *Trojan* categories, but in less than 40% of request groups.

It must be noted that non-ASCII characters are rarely seen in the browser benign HTTP request payloads, but they are present in the traffic of all browsers. Only for the Firefox and Chrome browsers on Windows 7 OS, the numbers are higher than 1% (3.40% and 1.26%, respectively). After performing manual analysis of these requests, it turned out that these were either a part of a JSON with Chinese UTF encoded characters or a part of data sent to URL: http://sqm.microsoft.com/sqm/vstudio/sqmserver.dll.

To summarize, this feature should be considered useful for identifying malicious HTTP traffic.

5.3.3. Methods of Requests with Payload. Request methods with payload data in the malware traffic were also analyzed. The majority of malware categories used *POST* request to send data. However, 8 malware categories used requests other than *POST*. In case of *Clicker* and *Spambot*, *GET* requests contained payload in 25.00% and 46.67% of request groups, respectively. In the remaining 6 categories, mixed values were present—*GET* or *POST* requests could both be present in request groups. They were also mixed with requests without payload.

Analogous analysis has been performed on the benign browser traffic. It turned out that all requests used *POST* methods.

Therefore, the comparison of the results for both types of traffic leads to the conclusion that in some cases, the feature can be used to distinguish between these two traffic types. Note that RFC 7231 does not prohibit sending payload data in the *GET* requests; however, as it can be seen from the obtained results, browsers usually perform such operation using the *POST* method.

5.3.4. Presence of Referer Header in POST Requests. Finally, the presence of the *Referer* header in *POST* requests of malicious HTTP traffic has been investigated.

The obtained results show that most malware categories sent *POST* requests without the *Referer* header. However, in the *Ransomware* category, this header was present in almost 55% of request groups. Only in 2 categories (i.e., *Stealer* and *Bruteforce*) *Referer* was spotted in more than 10% request of groups (27.78% and 14.29%, respectively).

POST requests constitute a small fraction of all browser requests in the analyzed traffic, i.e., less than 1.5% depending on the browser. For 2 browsers, the *Referer* header is present in every *POST* request (Internet Explorer 11 and Firefox with Flash Player, both on Windows 7). In the other case, only at most in 3.17% of all *POST* requests, this header is not present.

Based on the comparison of the obtained results for malware and browser traffic, it can be concluded that the lack of presence of the *Referer* header in *POST* requests can be a promising feature to distinguish malicious and benign HTTP traffic.

5.4. Comparison of Results with the Related Work. In this section, the obtained results are compared with those reported in the previously discussed papers (see Section 2). In the remainder of this section, only sources which explored features of HTTP network traffic are taken into account, as these can be directly compared to this work.

In two sources (Rossow et al. in [11] and Nelson in [12]), a number of unique header names in malware traffic was observed. In the first source, it was 144 headers, and in the second one, it was 24, whereas in this analysis, it was 42. Also, as reported by Rossow et al. [11], in 98.6% of samples the *User-Agent* header was present. The below paper revealed that depending on the malware category, it was at least 57.78%, but in many cases, the percentage was closer to 80%. Rossow et al. observed the *Accept-Language* header in 44.3% of samples—in this analysis, this header was hardly present in requests at all. Nelson in [12] also analyzed the *Accept-Language* header along with *Content-Type*, considering values of these headers as helpful for identifying malware. In this paper, their values were not analyzed extensively, but it was observed that their presence (or lack) can be used as a distinctive feature to spot malware traffic.

Calzarossa et al. in [13] analyzed benign HTTP network traffic. The authors observed HTTP/1.0 version of protocol in 4% of requests, whereas in the below research, it was not observed at all or it was present in less than 0.1% of requests. Also, they reported about 60 unique header names with their number between 0 and 14 in request. In the below analysis, it was about 90 unique header names, with 0 to 24 headers in a single request. The most popular headers were similar in both analyses, apart the *From* header which was not present in the below dataset. It can be explained with the nature of analyzed network traffic. This header is popular in requests sent by robot HTTP clients, and such traffic was present in the discussed analysis. However, it was not present in the traffic analyzed here.

In Section 2, nonacademic sources were also reviewed. Montoro in his presentation [14] presented a set of request features, which can be used for malware detection. Some

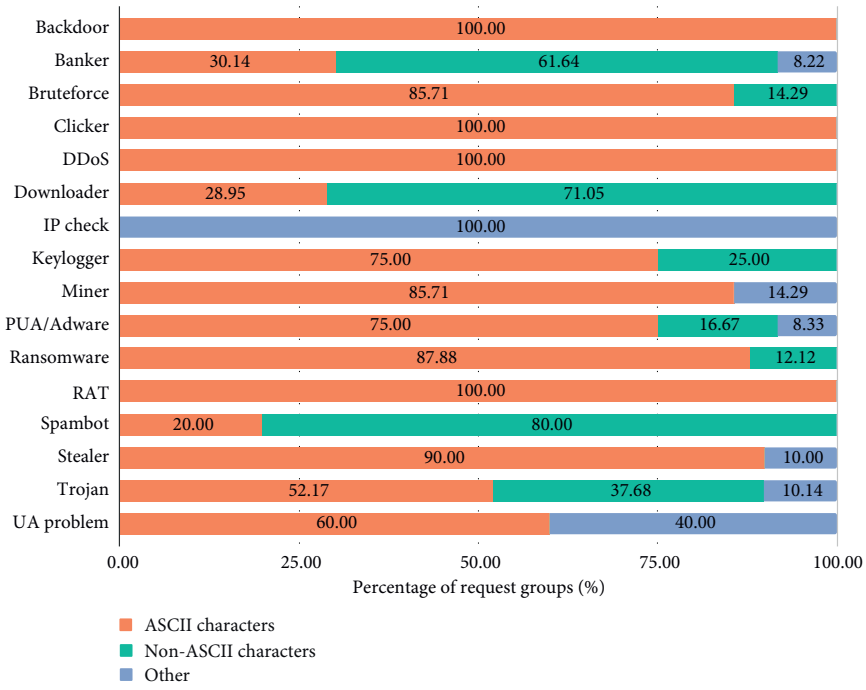


Figure 10: Non-ASCII characters in the payload (malicious traffic).

of them were also identified here as distinctive for malware network traffic. These include lack of popular requests, protocol version, and the number of headers in request. The author observed that malware sometimes does not include the *User-Agent* header; this was also observed in the below research. Additionally, he also identified that malware sometimes sends less than 4 headers, while benign applications send usually more than 9. In the below analysis, a similar dichotomy was observed; i.e., malware tends to send less headers in request than browsers. However, the numbers were different, as malware tends to send at most 6 headers and browsers between 7 and 9.

Analysis included in community modules of the Cuckoo sandbox system (https://cuckoosandbox.org/) was also explored in this work. The features of HTTP traffic analyzed by the *network_cnc_http* module were also identified here, proving their usefulness. Lack of the *Referer* header in malware in *POST* requests was observed as well as the lack of the *User-Agent* header in *POST* and *GET* requests. Also, it was identified that HTTP/1.0 version of the protocol is often seen in malware requests. Regarding the module's feature of the IP address in the *Host* header value, it was observed that values other than the domain name are hardly seen (less than 1% of requests) in the in the browser traffic. The IP address is not as a popular *Host* value as the domain in the malicious dataset; however, it is still more frequent than in the browser traffic. As the value of the *Host* header depends on the infrastructure of the attacker, it can be used as a potential indicator in malicious traffic identification.

Lewis presented in [17] observations about HTTP headers sent by malware. The below analysis has confirmed the findings of the author that malware sometimes uses nonstandard values of *User-Agent*. However, the below experiments did not find frequent typographic errors in the header names and values. For example, the only features regarding the whitespace character which give any results were when the space was present before CRLF (present in the network traffic of 5 malware categories, only 2 categories exceeding 10%) and when the space was present before a comma (2.35% of request groups in the *Ransomware* category). Additionally, the double space was not present in the malware traffic, but it was observed in the browser traffic (below 0.1% of requests).

The differences in results between the below analysis and the reviewed papers can be explained as follows. Firstly, the datasets analyzed in all sources and in this analysis are not uniform; that is, the uniformity of represented malware families and samples is not guaranteed. Secondly, the reviewed work is older than this analysis, and some malware families' behavior could already change, for example, to be able to further avoid detection. This shows that the analysis of malware network behavior should be performed regularly, especially when observed against the behavior of benign software. In this case, having a continuous monitoring can significantly increase the chance of detection of evolving threats.

6. Application of the Conducted Research

Until this point, HTTP protocol requests were analyzed in order to identify features, which could be helpful in distinguishing between malicious (malware) and benign

(browser) HTTP traffic. In this section, the obtained results will be summarized to provide more practical and operational information and insights.

6.1. Practical Observations. In the previous analyses, the number of features indicated significant differences between the malware and browser traffic. These features are summarized in Table 18. Features marked with * were proposed by the authors at the beginning of this paper as worth of analysis. Please note that the results for the destination port other than 80 are limited; however, they are analyzed against values of the *Host* header.

The values for the number of headers were chosen according to analyses conducted in Section 5.1.2 which showed that only in a small number of browser requests, less than 4 headers were present. Also, the analyses indicated that the boundary of 6 headers in a request can be chosen as a distinctive value for the majority of requests between malicious and benign HTTP traffic.

High payload entropy is defined as greater than 6.13 bits. This specific value was chosen as the maximum of the entropy value observed in the browser traffic (Section 5.3.1). Such a definition can also be supported from the practical perspective as in the popular network tool CyberChef (https://gchq.github.io/CyberChef/#recipe=Entropy()) in which authors state that English texts' entropy value lies usually between 3.50 and 5 bits. Also, an analysis of the Zeus botnet by Al-Bataineh and White [35] showed that the payload entropy was higher than 6.5 bits which is similar to the value proposed in this paper.

The features presented in Table 18 were further analyzed to determine how often their pairs co-occur. The results of this analysis provide some practical observations. They can be used as indicators in the manual malware analysis or treated as an entry point for further analyses. The most important observations from co-occurrence of features analysis in the malicious HTTP requests are that in the significant number of requests:

(i) A low number of headers occurs with the lack of the *User-Agent* header

(ii) Requests with the high entropy payload do not have a domain in the *Host* header value or the requests use the *POST* method without the *Referer* header

(iii) When the request is sent to port other than 80, the *User-Agent* header value is different from the standard ones

(iv) When the *GET* request has payload, it also has a low number of headers or the entropy of payload is high or the payload contains non-ASCII characters

(v) The *POST* requests without the *Referer* header also have non-ASCII characters in payload

(vi) Requests without the *Accept* header also lack *Accept-Encoding* or *Accept-Language* headers

(vii) Requests without the *Connection* header also lack *Accept*, *Accept-Encoding*, or *Accept-Language* headers

(viii) When the request is sent to the port other than 80, the *Host* header value is not a domain

(ix) With *1.0* version of the protocol, *POST* requests do not contain the *Referer* header

6.2. Practical Usage Scenarios. HTTP request features presented in previous sections can be practically applied in multiple scenarios. The main ones are outlined below.

Firstly, some of the features, especially those presented in Section 6.1, can be used directly to identify suspicious requests. The term suspicious is used intentionally because the presence or lack of some of these features cannot be treated as an unambiguous indicator of the request maliciousness. Nevertheless, multiple usage scenarios can be presented. One of them is a manual inspection of the HTTP traffic during malware sample analysis. Also, the observations presented in this paper can be incorporated as rules in network security monitoring systems. Examples of such rules are presented in Figures 11 and 12. Such rules were used with success in the malware analysis laboratory of CERT Polska.

Secondly, all presented features can be used to create an application fingerprinting system. Such a system can create a unique identifier by extracting and investigating particular features of the HTTP traffic. The identifier can be attributed to the particular application and afterwards used as a pattern to recognize such application's network traffic. Fingerprinting systems are used for some protocols, for example, for the TLS with the JA3 system (https://github.com/salesforce/ja3) or HTTP with p0f (http://lcamtuf.coredump.cx/p0f3/). The latter is not actively developed; however, it can be used as an inspiration. The HTTP request fingerprinting system can be used to identify particular malware families but potentially can also help to reveal information about the nature or the purpose of the inspected requests. For example, it can provide information whether request was a C&C server beacon or a connectivity check. Also, in the strictly controlled environments, a list of allowed application fingerprints can be used, and if a fingerprint previously not encountered is detected, the system can raise an alarm. Observations presented in this paper were used to create a prototype of the HTTP analysis and fingerprinting module for the Long-Term Sandboxing subsystem in the Horizon 2020 SISSDEN Project [36]. The system helped in the observation of malware behavior, for example, by providing a means for identification of malware operations such as connecting to the C&C server or connectivity checks.

Thirdly, the presented analyses identified HTTP request features which allow to distinguish malicious and benign HTTP traffic. Such features can be conveniently used to create a malware detection system which utilizes them to provide information about maliciousness of the HTTP requests. As a result, information on whether infected hosts are present in the monitored network can be provided.

Table 18: Features indicating significant differences between malware and browser traffic.

Name of the feature
HTTP/1.0 version of protocol
0–3 headers
High entropy of the payload
Lack of the *User-Agent* header
Nonstandard value of the *User-Agent* header
Non-ASCII characters in payload*
Presence of POST request without the *Referer* header
Presence of GET request with payload*
Host header value other than domain*
Destination port other than 80
Lack of any of *Accept*, *Accept-Encoding*, *Accept-Language*, *Referer*, *Connection* headers

Features marked with * (an asterisk) were proposed originally by the authors at the beginning of this paper.

```
alert http any any -> any any \
(msg:"Suspicious POST request"; \
flow:established,to_server; content:"POST"; http_method; \
http_protocol; content:"HTTP/1.0"; \
http_header_names; content:!"Referer"; \
classtype:trojan-activity; sid:9000004;)
```

Figure 11: An example of the Suricata IDS rule based on presented observations. The rule detects *POST* requests in 1.0 version of the protocol without the *Referer* header.

```
@load policy/protocols/http/header-names.zeek

module HTTP;

export {
  redef enum Notice::Type += {
    Suspicious_HTTP_Request,
  };
}

event http_reply(c: connection, version: string,
                 code: count, reason: string){
  if ( |c$http$client_header_names| < 4 ){
    if (! c$http?$user_agent){
      NOTICE([$note=HTTP::Suspicious_HTTP_Request,
              $msg=fmt("Suspicious HTTP request"),
              $conn=c,
              $identifier=cat(c$id$resp_h)]);
    }
  }
}
```

Figure 12: An example of the Zeek network security monitoring analysis module based on presented observations. The rule detects requests with the number of headers less than 4 and without the *User-Agent* header.

7. Limitations of the Work

Even after carefully designed and performed analyses, some limitations of this research were identified. They are discussed below.

First of all, the presented generalized observations can be applied only to the analyzed malware samples. Other malware families or even other samples of the presented families can behave in a different way. The authors believe, however, that most of the identified HTTP request features can be generalized to other malicious software representatives as many of them capture differences inherent to the general malware behavior.

Secondly, every work aimed at describing general behavior directly depends on the quality of analyzed data, especially in the representation and identification of actual malware behavior. Much effort was put in providing high-quality and relevant data. Nevertheless, the quality of the malware execution process in the sandbox systems cannot be guaranteed, as these were not designed and maintained by the authors of this paper. It must be noted that some malware families are able to detect that they are being analyzed in a virtual environment, what triggers their different behavior or even termination of operations [37, 38]. As such, it could alter the analysis results. Yet, checking the presence of such antianalysis techniques could be a broad task and hard to perform using only pcap files without the knowledge of the machine-level behavior. In such a situation, it was assumed that the network traffic alerted by IDS will represent actual behavior and that the lack of the traffic will result in termination of analysis of a particular malware sample.

Thirdly, the analysis environment has an impact on the obtained results. The authors of this paper used industry-proven IDS rule sets as the ground truth to conduct the process of malware request detection and identification of their family names. The authors believe that it is a high-quality source of information, but as with every detection system, it is possible that some HTTP requests were detected mistakenly or were not detected at all. The detection of such cases would have required additional detection systems, which would have, in turn, introduced additional complexity of the system and would be partially in conflict with the rule sets' licenses.

Fourthly, malware request grouping and categorization could potentially introduce bias of data. As discussed in Section 4.2, the authors introduced such an approach to limit the impact of different sizes of malicious request sets. As an alternative approach, reducing the number of requests for some malware families was considered. However, the authors did not want to miss too much information related to the malicious software behavior within omitted requests. Also, as the reduction of a number of requests was feasible, addition of requests to some families to equate the numbers was not. Thus, such a reduction would also introduce data bias. From this perspective, the authors believe that their approach was more adequate, despite some data bias residues.

Fifthly, identified features are based on observed behavior differences of malware and browser network traffic. In a situation when malware is equipped with mimicking mechanisms, that is, its behavior is deliberately changed to imitate a browser, the features will deteriorate with regard to distinction of malicious and benign traffic, depending on the changed feature. The identified features have different levels of technical difficulty to introduce mimicking behavior, some of them deeply connected with the C&C protocol design, as with the payload entropy higher than 6 bits. Changing such behavior would require, for example, rejection of payload obfuscation or using other protocol for data exchange. The authors of this paper believe that in the majority of examples, simple mimicry techniques would be used, such as the changing value of the *User-Agent* header or adding the additional header, which still could not counter all identified features. In the authors' opinion, in an extreme scenario of nearly perfect imitation of browser traffic, malware would still manifest some features of network traffic which would differentiate it from the web browser, as these two types of software are designed to perform different tasks. However, such scenarios are out of scope of this study.

Sixthly, in this paper the authors focused on analysis of differences between malware and browser traffic from the perspective of network monitoring in the sandbox environment. This perspective can be changed to a centralized one using logs from an actual proxy server as the data source, where data would contain more diverse sets of HTTP clients. Such a change could have impact on results of the analysis.

Finally, malware can use the HTTPS protocol for communication; however, this study did not analyze such cases. As mentioned in the introduction, the HTTP protocol is more popular than HTTPS when used by malware, thus the authors have focused solely on it. Nevertheless, if HTTPS malware traffic is decrypted to the HTTP protocol (for example, in a sandbox environment), it can be analyzed using the identified features. It must be noted that the presented analysis did not utilize such decrypted network traffic; however, the authors believe it should be similar to nonencrypted traffic. In a scenario when HTTPS traffic cannot be decrypted, the below findings cannot be applied, and analysts should refer to techniques based on this protocol.

8. Conclusion and Future Work

This paper focuses on presenting extensive and systematic analyses of the HTTP requests for malware- and browser-generated traffic. Its main aim was to establish the most promising distinctive features which can be used to identify malicious requests. Several features have been designed based on the previous works and own experience from malware behavior analysis. Datasets of malware and browser network traffic were analyzed using these features to identify which can be utilized to distinguish between malicious and benign HTTP traffic. The obtained results indicate which features can be generally used to spot anomalies understood as a deviation from the normal behavior. It was identified that these features include HTTP/1.0 protocol version, number of headers smaller than 4, the lack of *Accept*, *Accept-Encoding*, *Accept-Language*, *Connection*, and to some extent *Referer* headers, the payload entropy higher than 6 bits, the occurrence of non-ASCII characters in the payload, and the presence of the *Referer* header in the *POST* requests.

A special category of features are those connected to the *Host* and *User-Agent* headers. Because of their purpose, their values are changed frequently. *Host* header values other than domain names, such as IP addresses, are more often experienced in the malicious than in the benign HTTP traffic. However, these values are strictly connected to the network infrastructure used by criminals and as such should be used in a controlled manner. In case of the *User-Agent* header, its value presents an even more complicated matter. Certainly, the lack of this header should be treated as an anomaly, but an analysis of its values is a more demanding task. The results in this paper indicate that many malware categories use well-known values, similar to those sent by the browsers. Nevertheless, a significant number of malicious software families use values which were not recognized as popular. Interestingly, many malware categories use predominantly one *User-Agent* header value.

Other analyzed features did not yield any significant results; i.e., these features were not observed in the traffic at all or were too scarce to be treated as deviations from the typical browser traffic. Some of these features could be seen as anomalies on their own, even without comparing them with those from the browser traffic because they break RFCs or standardization. Good examples are the lack of a colon in the header field, misspellings of the header name, requests sent to the TCP protocol port other than registered for HTTP, not ASCII printable character in the header value, a new line character other than CRLF, repetition of headers (applicable to the majority of them), or nonstandard whitespace characters in the header field (other than space or horizontal tabulator).

Results presented in this paper showed that nonacademic sources reviewed at the beginning of the analysis provide features which are helpful for distinction between malware and browser traffic. Only the category of typographic errors, presented by Lewis in [17], did not yield any results, as these errors were very rare or nonexistent in the analyzed datasets.

Some of the features need additional investigations. These include payload data length or the value of some popular headers such as *Accept*, *Accept-Encoding*, *Language*, and *Connection*. Analysis of their values could be correlated with the inspection of the *User-Agent* value in such way that the occurrence of the particular value in 1 header should imply a defined value in the other.

Additional issue is caused by the *GET* requests with payload data. They were not present in the analyzed browser dataset, but they were rarely seen in the malware dataset. RFC 7230 does not prohibit sending such requests, but experience tells to monitor them. Despite low-level occurrence of these requests, the authors consider them as an anomaly.

In order to search for suspicious requests, features and anomalies identified in the course of this analysis can be directly applied to the existing network monitoring systems, such as IDSs or malware sandboxes. Also, with the use of the presented results, it would be feasible to create a malware detection system. Such a system could detect new malware samples in which presented anomalies appear. Finally, this work can be utilized to create a fingerprinting system which can be used as an identification mechanism of similar malware requests or as a source to create a whitelist of known applications in the network. The authors plan to explore these directions in their future work.

Data Availability

A part of the pcap files used in this study origins from the Malware Capture Facility Project (MCFP) and is publicly available at https://www.stratosphereips.org/datasets-malware. PCAP files from CERT Polska's sandbox system and web browser traffic have not been made publicly available because of commercial confidentiality and privacy reasons.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this article.

Acknowledgments

This research was partially supported by the EU Horizon 2020 program towards the Internet of Radio-Light project (H2020-ICT 761992).

References

[1] M. Trevisan, D. Giordano, I. Drago, M. Mellia, and M. Munafo, "Five years at the edge: watching Internet from the ISP network," in *Proceedings of the 14th International Conference on emerging Networking Experiments and Technologies*, ACM, Heraklion, Greece, pp. 1–12, December 2018.

[2] P. Richter, N. Chatzis, G. Smaragdakis, A. Feldmann, and W. Willinger, "Distilling the Internet's application mix from packet-sampled traffic," in *Passive and Active Measurement*, J. Mirkovic and Y. Liu, Eds., pp. 179–192, Springer International Publishing, Cham, Switzerland, 2015.

[3] S. Miller and P. Smith, *Rise of Legitimate Services for Backdoor Command and Control*, Anomali, Tech. Rep., 2017, https://www.anomali.com/files/anomali-labs-reports/legit-services.pdf.

[4] S. Tkachenko, *Stop Windows 10 Spying on You Using Just Windows Firewall*, https://winaero.com/blog/stop-windows-10-spying-on-you-using-just-windows-firewall/, 2015.

[5] Block Windows Update with Firewall, 2018, https://www.reddit.com/r/MoneroMining/comments/8l5wpt/block_windows_update_with_firewall/.

[6] B. Duncan, *2019-03-15-Malspam Pushes Lokibot*, http://malware-traffic-analysis.net/2019/03/15/index2.html, 2019.

[7] P. Srokosz, *Analysis of Emotet V4*, https://www.cert.pl/en/news/single/analysis-of-emotet-v4/, 2017.

[8] E. Brumaghin and H. Unterbrink, *Picking Apart Remcos Botnet-In-A-Box*, https://blog.talosintelligence.com/2018/08/picking-apart-remcos.html, 2018.

[9] R. Joven, *New Stealth Worker Campaign Creates a Multi-Platform Army of Brute Forcers*, https://www.fortinet.com/blog/threat-research/new-stealth-worker-campaign-creates-a-multi-platform-army-of-bru.html, 2019.

[10] A.-T. GmbH, *AV-TEST Security Report 2018/2019*, AV-TEST Institute, Tech. Rep., 2019, https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2018-2019.pdf.

[11] C. Rossow, C. J. Dietrich, H. Bos et al., "Sandnet: network traffic analysis of malicious software," in *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, ACM, New York, NY, USA, pp. 78–88, 2011.

[12] A. Nelson, "Sandnet++-a framework for analysing and visualising network traffic from malware," *Information Security Group*, Royal Holloway University of London, Tech. Rep., 2016.

[13] M. C. Calzarossa and L. Massari, "Analysis of header usage patterns of HTTP request messages,," in *Proceedings of the 2014 IEEE International Conference on High Performance Computing and Communications, 2014 IEEE 6th International Symposium on Cyberspace Safety and Security, 2014 IEEE 11th International Conference on Embedded Software and Syst (HPCC, CSS, ICESS)*, pp. 847–853, IEEE, Paris, France, 2014.

[14] R. Montoro, "HTTP Header Hunter-Looking for malicious behavior into your HTTP header traffic," 2011, http://2011.video.sector.ca/video/39786962.

[15] "Cuckoo sandbox network CnC HTTP community module," 2019, https://github.com/cuckoosandbox/community/blob/master/modules/signatures/network/network_cnc_http.py.

[16] "Cuckoo sandbox multiple user-agents community module," https://github.com/cuckoosandbox/community/blob/master/modules/signatures/windows/multiple_ua.py.

[17] T. Lewis, "HTTP header heuristics for malware detection," SANS Institute InfoSec Reading Room, Tech. Rep., 2013.

[18] S. Mizuno, M. Hatada, T. Mori, and S. Goto, "Botdetector: a robust and scalable approach toward detecting malware-infected devices," in *Proceedings of the 2017 IEEE Int. Conference on Communications (ICC)*, IEEE, Paris, France, pp. 1–7, May 2017.

[19] Z. Li, L. Sun, Q. Yan, W. Srisa-an, and Z. Chen, "Droidclassifier: efficient adaptive mining of application-layer header for classifying android malware," in *Proceedings of the International Conference on Security and Privacy in Communication Systems*, Springer, Guangzhou, China, pp. 597–616, October 2016.

[20] N. Kheir, "Behavioral classification and detection of malware through HTTP user agent anomalies," *Journal of Information Security and Applications*, vol. 18, no. 1, pp. 2–13, 2013.

[21] K. Li, R. Chen, L. Gu, C. Liu, and J. Yin, "A method based on statistical characteristics for detection malware requests in network traffic," in *Proceedings of the 2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, IEEE, Guangdong, China, pp. 527–532, June 2018.

[22] R. Perdisci, W. Lee, and N. Feamster, "Behavioral clustering of HTTP-based malware and signature generation using malicious network traces," in *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*, p. 14, San Jose, CA, USA, 2010.

[23] M. Mimura and H. Tanaka, "Leaving all proxy server logs to paragraph vector," *Journal of Information Processing*, vol. 26, pp. 804–812, 2018.

[24] M. A. Nia, R. E. Atani, B. Fabian, and E. Babulak, "On detecting unidentified network traffic using pattern-based random walk," *Security and Communication Networks*, vol. 9, no. 16, pp. 3509–3526, 2016.

[25] R. Fielding, J. Gettys, J. Mogul et al., *RFC 2616: Hypertext Transfer Protocol–HTTP/1.1*, 1999.

[26] R. Fielding and J. Reschke, *RFC 7230: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*, 2014.

[27] RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, 2014.

[28] RFC 7232: Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests, 2014.

[29] R. Fielding, Y. Lafon, and J. Reschke, *RFC 7233: Hypertext Transfer Protocol (HTTP/1.1): Range Requests*, 2014.

[30] R. Fielding, M. Nottingham, and J. Reschke, *RFC 7234: Hypertext Transfer Protocol (HTTP/1.1): Caching*, 2018.

[31] R. Fielding and J. Reschke, *RFC 7235: Hypertext Transfer Protocol (HTTP/1.1): Authentication*, 2014.

[32] T. Berners-Lee, R. Fielding, and H. Frystyk, *RFC 1945: Hypertext Transfer Protocol–HTTP/1.0*, 1996.

[33] C. Linhart, A. Klein, R. Heled, and S. Orrin, *HTTP Request Smuggling*, Watchfire Corporation, Tech. Rep., 2005.

[34] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, *RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, 2008.

[35] R. R. Scherberger, H. Kaess, and S. Brückner, "Studies on the action of an anticholinergic agent in combination with a tranquilizer on gastric juice secretion in man," in *Proceedings of the 2012 7th International Conference on Malicious and Unwanted Software*, IEEE, Washington, DC, USA, pp. 1460–1463, October 2012.

[36] SISSDEN, *Secure Information Sharing Sensor Delivery Event Network (SISSDEN). Deliverable D5.3: Final Data Analysis Results*, https://sissden.eu/download/SISSDEN-D5.3-Final_Data_Analysis_Results.pdf, 2019.

[37] R. R. Branco, G. N. Barbosa, and P. D. Neto, *Scientific but not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies*, 2012.

[38] P. Chen, C. Huygens, L. Desmet, and W. Joosen, "Advanced or not? a comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware," in *Proceedings of the IFIP International Information Security and Privacy Conference*, Springer, Ghent, Belgium, pp. 323–336, May 2016.

5 Identyfikacja żądań złośliwego oprogramowania

By móc identyfikować żądania HTTP, tzn. by je precyzyjnie oznaczać i odróżniać, stosuje się różne rozwiązania tworzące ich tzw. *odcisk palca* (fingerprint). W działaniu mechanizm ten jest podobny do obliczania funkcji skrótu (np. SHA-256) z pliku, dzięki czemu można identyfikować pojedyncze pliki, a tym samym je odróżniać. Dzięki takiej identyfikacji, możliwe staje się śledzenie komunikacji złośliwego oprogramowania, rozpoznawania rodzajów komunikacji (np. komunikacja z serwerem C&C, atak typu DDoS lub click fraud), a także rozpoznawanie różnych rodzin złośliwego oprogramowania. Obecne narzędzia służące do tworzenia odcisków żądań HTTP, takie jak np. FATT [21], p0f [22] lub Mercury [23] nie analizują całości informacji dostępnej w żądaniach, lub robią to w pobieżny sposób, np. nie analizują danych przesyłanych w żądaniu (tzw. payload).

Ze względu na powyższe problemy, stworzono narzędzie *Hfinger*, który dokonuje analizy żądań HTTP złośliwego oprogramowania w celu stworzenia ich unikalnej reprezentacji. Reprezentacja ta pozwala na identyfikację tego typu żądań, także w sieciach z ruchem sieciowym aplikacji niezłośliwych. Hfinger analizuje ścieżkę URI, wersję protokołu, użytą metodę, nagłówki żądania i ich wartości oraz dane przesyłane w żądaniu. Umożliwia to stworzenie krótkiego łańcucha znakowego, którego składowe mogą być w łatwy sposób interpretowane przez analityka. Hfinger skupia się na analizie żądań protokołu HTTP ze względu na fakt, że w niektórych sytuacjach użycia narzędzia odpowiedzi serwerów mogą nie być dostępne lub nie są poprawne. Może mieć to miejsce w sytuacji, gdy serwer C&C już nie działa lub został przejęty przez badaczy bezpieczeństwa (sinkholing) i odpowiedzi nie są już takie same jak przy oryginalnej infrastrukturze. Z kolei położenie nacisku na badania w środowisku aplikacji systemu operacyjnego Windows spowodowane jest liczbą próbek złośliwego oprogramowania, która jest nadal większa niż dla innych systemów operacyjnych[24].

Głównym wkładem pracy badawczej w tym rozdziale jest:

- Przeprowadzenie przeglądu i analizy popularnych narzędzi umożliwiających stworzenie odcisku protokołu HTTP.
- Przedstawienie nowego narzędzia do tworzenia odcisków żądań HTTP złośliwego opro-

[21] `https://github.com/0x4D31/fatt`
[22] `https://lcamtuf.coredump.cx/p0f3`
[23] `https://github.com/cisco/mercury`
[24] `https://portal.av-atlas.org/malware/statistics` Dostęp: 10.12.2021 r.

gramowania.

- Przeprowadzenie analizy zaproponowanego narzędzia i porównaniu jego efektywności do innych, popularnych narzędzi umożliwiających stworzenie odcisku żądań protokołu HTTP.

Mimo że Hfinger nie jest detektorem, tzn. nie wykrywa i nie klasyfikuje złośliwego oprogramowania, odciski żądań HTTP przez niego tworzone mogą być oznaczane przy użyciu np. ręcznej analizy lub systemów detekcji włamań sieciowych. Dzięki takiemu rozszerzeniu mechanizmu, Hfinger może umożliwić detekcję malware'u.

Pracy badawcza nad charakterem ruchu sieciowego protokołu HTTP złośliwego oprogramowania, zaprezentowana w rozdziale 4, umożliwiła sformułowanie i skonstruowanie 14 cech żądań HTTP, których zadaniem jest stworzenie unikalnej reprezentacji żądań złośliwego oprogramowania. Te cechy to:

1. Długość ścieżki URI.
2. Liczba poziomów katalogów w URI.
3. Średnia długość katalogu URI.
4. Rozszerzenie żądanego pliku.
5. Długość części ze zmiennymi w URI.
6. Liczba zmiennych w URI.
7. Średnia długość wartości zmiennych URI.
8. Metoda żądania.
9. Wersja protokołu.
10. Reprezentacja kolejności nagłówków żądania.
11. Reprezentacja wartości popularnych nagłówków.
12. Obecność znaków spoza zbioru ASCII w przesyłanych danych.
13. Entropia Shannona przesyłanych danych.

14. Rozmiar przesyłanych danych.

Efektywność Hfingera została przeanalizowana w porównaniu do trzech wcześniej wspomnianych narzędzi: FATT, p0f i Mercury. Bazowano przy tym na czterech miarach:

- *Poziomie kolizji dla złośliwego oprogramowania*, określającym częstość, z jaką odciski jednej rodziny złośliwego oprogramowania występują w innych rodzinach. Liczba ta powinna być jak najmniejsza, zapewniając unikalność reprezentacji.

- *Poziomie liczby generowanych odcisków*, określającym jak wiele odcisków tworzonych jest średnio na żądanie HTTP. Liczba ta powinna być jak najmniejsza, co świadczyłoby o skutecznym grupowaniu żądań.

- *Poziomie kolizji z aplikacjami niezłośliwymi*, określającym częstość, z jaką odciski złośliwego oprogramowania są takie same jak dla oprogramowania niezłośliwego. Liczba ta powinna być jak najmniejsza, zapewniając unikalność reprezentacji żądań także w obecności ruchu sieciowego aplikacji niezłośliwych.

- *Entropii*, bazującej na entropii Shannona, informującej o średniej ilości informacji zawartej w odciskach. Liczba ta powinna być jak najwyższa.

Badania zostały przeprowadzone na zbiorach danych żądań HTTP złośliwego oprogramowania oraz popularnych niezłośliwych aplikacji. Zbiór danych z ruchem sieciowym złośliwego oprogramowania obejmował 121 rodzin oraz 401 566 żądań HTTP. Został stworzony na bazie połączenia danych z wcześniejszych badań (por. rozdział 4.) oraz danych pochodzących ze środowiska automatycznej analizy złośliwego oprogramowania zespołu CERT Polska, które zostały przygotowane specjalnie do analizy narzędzia Hfinger. Zbiór danych niezłośliwych aplikacji składał się z dwóch części: ruchu sieciowego popularnych przeglądarek obejmującego 194 940 żądań (pochodził z wcześniejszych badań) oraz specjalnie wygenerowanego ruchu sieciowego różnych popularnych aplikacji systemu Windows 10 i obejmował 53 717 żądań.

Analiza efektywności narzędzia Hfinger została podzielona na dwie części: wybór optymalnego zestawu cech składowych odcisku oraz właściwe porównanie efektywności z innymi narzędziami. Zbiór danych z żądaniami został podzielony losowo na dwie części: na jednej części został dokonany wybór cech, a na drugiej badania efektywności.

Wybór cech składowych odcisku miał za zadanie sprawdzenie jaki podzbiór 14 zdefiniowanych cech da najlepsze wyniki pod względem czterech zdefiniowanych miar, włączając w to

możliwość różnego zaokrąglenia wartości liczbowych niektórych z cech. Dla wszystkich spośród 186 623 różnych kombinacji cech obliczono wartości miar. Analiza wykazała, że dla wielu zestawów cech wraz ze wzrostem poziomu kolizji w złośliwym oprogramowaniu wzrastał poziom kolizji z aplikacjami niezłośliwymi, podczas gdy poziom liczby generowanych odcisków oraz ich entropia malały. Zgadza się to z intuicją, która podpowiada, że im odciski zawierają więcej informacji, tym mniej z nich koliduje między sobą, co jednak powoduje wzrost ich liczby. Ostatecznie po selekcji pozostało 5 zestawów cech, które zostały wybrane po zastosowaniu różnych celów optymalizacji, np. zminimalizowania poziomu kolizji dla złośliwego oprogramowania, ograniczeniu liczby generowanych odcisków lub maksymalizacji entropii.

Wszystkie wybrane zestawy cech zostały następnie porównane z trzema narzędziami oferującymi tworzenie odcisków dla żądań HTTP. Analiza wykazała, że Hfinger uzyskał od 8 do 34 razy niższe poziomy kolizji między rodzinami złośliwego oprogramowania niż pozostałe narzędzia. Dodatkowo w domyślnym trybie działania Hfinger nie wprowadza kolizji z aplikacjami niezłośliwymi, podczas gdy w innych narzędziach zdarzało się w od 10 do 30% przypadków. Lepsze wyniki zostały osiągnięte przy co najwyżej trzykrotnym zwiększeniu liczby generowanych odcisków.

Jak wykazały badania Hfinger umożliwia bardziej unikalną reprezentację żądań HTTP złośliwego oprogramowania niż inne popularne narzędzia. Dzięki temu może być używany do identyfikacji i śledzenia złośliwego oprogramowania. Typowe scenariusze użycia to grupowanie żądań ze względu na ich funkcję (np. połączenie z serwerem C&C lub test łączności z internetem), wyszukiwanie nowych żądań na bazie wcześniej przypisanych do konkretnych rodzin złośliwego oprogramowania, uzupełnienie systemu wykrywania włamań sieciowych przez odnajdywanie niezaraportowanych żądań o takim samym odcisku jak te wykryte przez system. Dzięki odpowiedniej konstrukcji reprezentacji informacji o żądaniu w odcisku, Hfinger może być używany zarówno w systemach automatycznych, np. monitorująco-logujących typu SIEM lub automatycznej analizy próbek (sandbox), ale także do ręcznej analizy przez analityków. Narzędzie zostało opublikowane w serwisie GitHub[25] i jest używane w środowisku automatycznej analizy złośliwego oprogramowania zespołu CERT Polska (realizującego zadania CSIRT NASK jako jednego z trzech zespołów CSIRT poziomu krajowego według ustawy o krajowym systemie cyberbezpieczeństwa).

Praca badawcza nad charakterem ruchu sieciowego protokołu HTTP złośliwego oprogramo-

[25] `https://github.com/CERT-Polska/hfinger`

wania, zaprezentowana w rozdziale 4 umożliwiła sformułowanie i konstrukcję cech żądań HTTP, na których bazuje Hfinger. Przełożyło się to na jego wysoką efektywność w identyfikacji żądań HTTP złośliwego oprogramowania. W rozdziale 6. dokonano analizy przydatności odcisków tworzonych przez Hfinger do klasyfikacji rodzin złośliwego oprogramowania.

Pełen opis badań nad narzędziem Hfinger został opublikowany pod tytułem *Hfinger: Malware HTTP Request Fingerprinting* w czasopiśmie *Entropy*, 2021, vol. 23, nr 5, s.1-25, Numer artykułu: 507. [12]. Kopia artykułu została umieszczona poniżej jako część rozdziału.

entropy

MDPI

Article

Hfinger: Malware HTTP Request Fingerprinting

Piotr Białczak [1,2,*] and Wojciech Mazurczyk [2]

[1] CERT Polska/Research and Academic Computer Network (NASK), Kolska 12, 01-045 Warsaw, Poland
[2] Institute of Computer Science, Warsaw University of Technology, Nowowiejska 15/19, 00-665 Warsaw, Poland; wojciech.mazurczyk@pw.edu.pl
* Correspondence: piotr.bialczak@cert.pl

Abstract: Malicious software utilizes HTTP protocol for communication purposes, creating network traffic that is hard to identify as it blends into the traffic generated by benign applications. To this aim, fingerprinting tools have been developed to help track and identify such traffic by providing a short representation of malicious HTTP requests. However, currently existing tools do not analyze all information included in the HTTP message or analyze it insufficiently. To address these issues, we propose Hfinger, a novel malware HTTP request fingerprinting tool. It extracts information from the parts of the request such as URI, protocol information, headers, and payload, providing a concise request representation that preserves the extracted information in a form interpretable by a human analyst. For the developed solution, we have performed an extensive experimental evaluation using real-world data sets and we also compared Hfinger with the most related and popular existing tools such as FATT, Mercury, and p0f. The conducted effectiveness analysis reveals that on average only 1.85% of requests fingerprinted by Hfinger collide between malware families, what is 8–34 times lower than existing tools. Moreover, unlike these tools, in default mode, Hfinger does not introduce collisions between malware and benign applications and achieves it by increasing the number of fingerprints by at most 3 times. As a result, Hfinger can effectively track and hunt malware by providing more unique fingerprints than other standard tools.

Keywords: fingerprinting; malware analysis; malicious network traffic analysis; HTTP protocol analysis; pcap file analysis; malware tracking; malware identification

check for updates

Citation: Białczak, P.; Mazurczyk, W. Hfinger: Malware HTTP Request Fingerprinting. *Entropy* **2021**, *23*, 507. https://doi.org/10.3390/e23050507

Academic Editor: Amelia Carolina Sparavigna

Received: 26 March 2021
Accepted: 19 April 2021
Published: 23 April 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Currently, malicious software (malware) developers use Hypertext Transfer Protocol (HTTP) as one of the primary carriers for malicious communication. According to Miller et al. [1], HTTP is the most common protocol used in the Command and Control (C&C) traffic, more popular than Hypertext Transfer Protocol Secure (HTTPS). It is utilized by malware, e.g., to connect to the C&C server to register or obtain commands, check the infected machine's IP address, or download additional modules. Moreover, it can be used to perform DDoS (Distributed Denial of Service) attacks or to click on referral links, thus creating revenue.

To identify and discern different malware communication activities, network traffic fingerprinting methods can be applied. The notions of a fingerprint and fingerprinting as the act of creating a fingerprint are similar to the notions of classic forensic work, where the fingerprint is an impression of human fingers' friction ridges. In the field of computer science, a working definition of a fingerprint is a short representation of a larger object [2]. The most crucial property of fingerprinting is that two different objects have different fingerprints, and the probability of a collision, i.e., an event when two different objects have the same fingerprint, is low. File fingerprinting is one of the application examples, where cryptographic hash functions, e.g., such as SHA-256, are used to create identification tags for the fingerprinted files.

However, it is not only files that can be fingerprinted. Network traffic can also be used for this purpose. Various network protocols can be analyzed to represent the exchanged data, which then, in turn, can be used for identification purposes. The process of fingerprinting can be conducted in an active or passive mode. The former is performed with a modification of the standard network traffic, for example, by sending carefully crafted messages. In the latter, no changes are introduced, and network traffic is only monitored. The most popular practical usage examples are passive Operating System fingerprinting (as realized, e.g., in p0f https://lcamtuf.coredump.cx/p0f3, accessed on 26 March 2021), web browser fingerprinting (like in, e.g., privacy research and advocacy service at https://panopticlick.eff.org/, accessed on 26 March 2021) or network service discovery (as performed by, e.g., Nmap https://nmap.org/, accessed on 26 March 2021). Network protocols can also be fingerprinted to identify, track, or detect malware. Until now, several such examples of network traffic fingerprinting methods and tools exist, and they are discussed in detail in Section 2.

From the variety of network protocols, HTTP fingerprinting is a promising approach to provide identification and tracking of malware communications, which is crucial for malware analysts in their daily work while defending networks. Currently, several tools have been proposed that help fingerprinting HTTP traffic, such as FATT (https://github.com/0x4D31/fatt, accessed on 26 March 2021), p0f (https://lcamtuf.coredump.cx/p0f3, accessed on 26 March 2021), or Mercury (https://github.com/cisco/mercury, accessed on 26 March 2021). However, they all share the same limitation. In our opinion, they do not analyze all information included in the HTTP messages or analyze it insufficiently. For example, the above-mentioned tools do not analyze the payload of the request, and the URI analysis is at most limited to value encoding. Note that both these features have already been proven to be useful for malware detection and identification purposes, for example, as described by Li et al. [3] or Perdisci et al. [4]. Fingerprints created with such an approach omit information that can potentially identify and discern various malware families' requests.

To address these issues, we propose the Hfinger tool that aims to fingerprint malware HTTP requests more comprehensively. In more detail, Hfinger processes HTTP requests and generates a fingerprint based on the URI, protocol version, the request method, headers and their values, and the request's payload. The tool's main goal is to produce unique representations of malware HTTP requests, thus providing a mechanism for the identification of such requests in network traffic of various applications. The fingerprint created by Hfinger represents a malware request in a short and concise form that can still be interpretable by a human analyst. Hfinger was designed to be used with exact match searching mechanisms, which provide means for direct fingerprint searching without using wildcard techniques. Exact match searching is supported by, for example, many security monitoring and logging tools. In this vein, it is intended to provide similar search functionality as malware sample hashes such as SHA-256. Nevertheless, its overt nature can still help analysts by giving basic information about the request or by finding patterns in the network traffic.

Hfinger does not provide direct identification of particular malware families or directly detect malware per se. However, it can identify requests that can be labeled as malicious using other information sources, such as Intrusion Detection Systems. The tool can also be used in threat hunting to uncover unknown requests that were omitted by other security solutions but that share fingerprint with those identified as malicious. The tool is open source and has been published at https://github.com/CERT-Polska/hfinger, accessed on 26 March 2021. The research presented in this paper is focused on HTTP requests of Windows-based malware. The utilization of only HTTP requests is related to the fact that the server responses may be unavailable in some of the fingerprinting tool usage scenarios. For example, when analyzing an old malware sample for which C&C servers no longer work, or server responses are unreliable because the malicious infrastructure was sinkholed and the received messages are different from the original. From this perspective, using only requests for fingerprinting purposes can be more reliable in analyzing malware's actual

behavior. Moreover, focusing on Windows-based malware is related to the fact that it is still the most frequently attacked platform despite the constant increase in the number of threats on mobile platforms. According to AV-TEST Institute in 2019, more than 75% of malware targeted Windows operating system [5].

To prove the effectiveness of the proposed Hfinger tool we have conducted an extensive experimental study. To this aim, we based the performed evaluation on four main metrics that measure fingerprint collision probability for various malware families (separately and including benign software), the number of created fingerprints, and their entropy. Moreover, to determine the performance of the proposed solution for malware HTTP request fingerprinting, we use real-world malware and benign applications data sets containing HTTP traffic. In more detail, the former consists of 121 popular malware families represented using 401,566 HTTP requests, while the latter incorporates 248,657 HTTP requests generated by popular Windows applications, including web browsers. Additionally, the effectiveness of the developed tool has been compared with the three existing, previously mentioned community-proven HTTP fingerprinting tools, i.e., FATT, p0f, and Mercury.

Considering the above, the main contributions of this paper are as follows:

- Proposing Hfinger—a new malware HTTP request fingerprinting tool;
- Performing a review and analysis of popular HTTP fingerprinting tools;
- Providing an extensive experimental evaluation of the proposed approach and its comparison with the popular, existing HTTP fingerprinting tools.

The rest of the paper is structured as follows. First, Section 2 describes the most notable related work. Then, in Section 3, we present the proposed HTTP fingerprinting tool. Next, Section 4 contains details on the chosen experimental methodology, while in Section 5 obtained results are included and discussed. Section 6 showcases potential practical usage scenarios for Hfinger and pinpoints its main limitations. Finally, Section 7 concludes our work and indicates potential future research directions.

2. Related Work

In this section, first we review the most important work related to the topic of this paper and compare it to Hfinger. Then we describe existing popular tools used for HTTP traffic fingerprinting and we discuss their limitations. Finally, we compare them to the solution proposed in this paper.

An important distinction between the presented research solutions and tools must be drawn. The former were created to provide an extensive research analysis of a particular problem which, in some cases, resulted in creation of a tool or a system that solves the stated problem. On the other hand, the latter were primarily focused on creation of a tool that solves a specific technical (rather than a research) problem and the tool's analysis is typically quite limited.

2.1. Proposed Research Solutions

Research on web browser fingerprinting is directly related to HTTP malware fingerprinting, and this topic has been extensively covered in the literature (cf. Laperdrix et al. [6]). This type of fingerprinting is based on active and passive techniques in which information about different features of the environment, web browser, and OS are extracted. While for active fingerprinting different techniques are used, such as JavaScript to query information about the canvas, a list of browser plugins, or screen resolution, passive fingerprinting techniques analyze requests sent by web browsers. Common techniques involve checking the values of popular headers such as *User-Agent*, *Accept*, or *Content-Encoding* but also headers' order. Hfinger utilizes these passive fingerprinting techniques; however, they are extended, for example, with URI and payload analysis.

Fingerprinting of HTTP network traffic can be used to create models of applications present in a monitored network and used as a baseline for detecting unknown applications that can be malicious. Bortolameotti et al. presented in [7] DECANTeR a system for

detection of HTTP network traffic that is anomalous for analyzed host. It passively extracts fingerprints of benign applications running on the host. This process involves extracting information from clustered *POST* and *GET* requests in the form of *Host* header value, constant header fields, average request size, *User-Agent* header value, *Accept-Language* header value, and the size of outgoing information in the cluster.

Bortolameotti et al. presented in [8] a system for the detection of anomalous traffic. Their system uses two models of header-value entropy and header sequence extracted from multiple requests to create known applications' fingerprints. After the training phase, the system can evaluate if an unknown request is similar to already known applications or it originates from a new application. Comparing these two approaches to Hfinger shows they were designed with different objectives than the latter. Their goal is to provide a model of application behavior based on multiple HTTP requests to create a baseline for detecting outlying applications in a particular network, thus providing malware detection. On the other hand, Hfinger is focused on the unique representation of malware HTTP requests, providing a fingerprint for each separate request. Such an approach allows analyzing network traffic without the baseline model creation stage and analyzing network traffic with single requests, for example, when the infrastructure of analyzed malware is not working anymore. Furthermore, while all these systems analyze similar parts of the requests, Hfinger utilizes a broader set of features for fingerprint generation and analyzes all requests, regardless of their method. In contrast, for example, DECANTeR fingerprints clustered *GET* and *POST* requests only.

Various approaches have been proposed for fingerprinting other popular network protocols. Many studies focus on HTTPS protocol, where the primary research objectives are HTTPS network traffic presence identification or identification of services utilizing HTTPS for communication (cf. [9]). SMTP network traffic fingerprinting can be used to identify malware families as presented in [10,11]. SMTP messages, SMTP extensions, and IMF fields are used to create different e-mail clients' dialects, thus providing a method for their identification. DNS protocol fingerprinting can be used as a method to detect DNS amplification DDoS attacks [12], identify DNS servers [13], or for the detection of bots [14]. Segal et al. in [15] presented a white paper on fingerprinting of HTTP/2 protocol clients. When used for malware network traffic fingerprinting, the presented approaches can be applied to identify malware families but also to detect some specific operations, such as sending spam messages or performing DDoS attacks.

Other approaches for network traffic fingerprinting with more generic methods exist, too. For example, Holland et al. in [16] proposed nPrint—a system for standard representation of network traffic. For every analyzed packet, its representation is created that maps all packet bytes to a feature vector representing all possible headers of a particular protocol. The authors claim that their system can generate data suitable as input for machine learning algorithms in classification problems. Unfortunately, when fingerprinting HTTP network traffic, nPrint needs to be configured with appropriate rules, defining which parts of the messages should be extracted. Therefore, it does not provide ready-made methods for HTTP fingerprinting.

Table 1 introduces the classification of the existing fingerprinting research based on its application scenario. It also shows that these approaches utilize fingerprinting for various purposes—some provide identification of benign services or applications, while the other uncover malicious activities and software.

Although the research solutions described above provide fingerprinting mechanisms of various network protocols, they differ from Hfinger in several aspects. Browser fingerprinting methods use, mainly, active analysis techniques and are focused on one type of HTTP clients. Hfinger utilizes only passive analysis techniques, and, despite the focus on malware requests, its design and performance analysis considered the presence of benign HTTP clients. DECANTeR and HeadPrint systems utilize multiple HTTP requests to create a baseline model of the observed network to identify requests that do not fit the created model, thus detecting unknown applications. Hfinger is focused on the unique

representation of single malware HTTP requests, which is a different research goal. nPrint provides generic fingerprints of various network protocols and requires the creation of configuration to produce HTTP request fingerprints. The configuration is not provided by the authors, so research into optimal HTTP representation is needed. Conversely, Hfinger is focused only on HTTP, for which a complete optimization has been performed. Segal et al. analyze HTTP in version 2 that is different than previous protocol versions in data transfer and representation techniques. Thus their analysis techniques cannot be directly compared to Hfinger. Other reviewed research solutions analyze protocols different than HTTP.

Table 1. Classification of the reviewed fingerprinting research solutions based on their application scenario.

Usage of Fingerprinting	Examples of Research Solution
Client application or user identification	Laperdrix et al. [6], Segal et al. [15]
Detection of unknown applications	Bortolameotti et al. [7,8]
Service or server identification	Shbair et al. [9], Kim et al. [13]
Malware family identification	Stringhini et al. [10], Bazydło et al. [11]
Malware detection	Blaise et al. [14]
Attack detection	Fachkha et al. [12]
Generic protocol fingerprinting	Holland et al. [16]

2.2. Existing HTTP Fingerprinting Tools

In this subsection, HTTP fingerprinting tools similar to the Hfinger are described. Three tools (FATT, Mercury, and p0f) have been selected based on capability of passive, pcap file based analysis of HTTP requests without any major code modification. Source code and documentation of these three applications are public, and they are well known in professional network security community.

Other tools fingerprinting HTTP do exist, but they use active fingerprinting for web browser identification (e.g., FingerprintJS—https://github.com/fingerprintjs/fingerprintjs, accessed on 26 March 2021) or they perform only server fingerprinting (httprecon—https://www.computec.ch/projekte/httprecon/, accessed on 26 March 2021, httprint—https://net-square.com/httprint.html, accessed on 26 March 2021, or nmap—https://nmap.org/, accessed on 26 March 2021). As such, they cannot be compared to the same extent to Hfinger as FATT, Mercury, or p0f.

When using the classification of fingerprinting research solutions presented in Table 1 the three reviewed tools, with some extensions, can be classified into groups providing identification of client applications, unknown applications, or malware families. All these tools produce fingerprints that, after the labeling process, can be used for searching the application defined by the labeled fingerprint. Conversely, if the created fingerprint does not fit a list of known applications, it can be used to identify an unknown application.

The three presented tools will be further analyzed and compared with Hfinger in Section 5. Details about their source code and configuration used in the analysis are presented in Section 4.3.

2.2.1. FATT

FATT—fingerprint all the things (https://github.com/0x4D31/fatt/, accessed on 26 March 2021)—is a tool for fingerprinting protocols such as SSL/TLS, SSH, RDP, HTTP, gQUIC. For HTTP, it provides means for fingerprinting headers of requests and responses by extracting header names into a list and computing MD5 hash from it. Depending on the chosen reporting format, the tool outputs additional contextual information, such as request URI, request full URI, request method and protocol version, and *User-Agent* value. However, these components are not used as a part of the fingerprint.

2.2.2. p0f

p0f—passive OS fingerprinting (https://lcamtuf.coredump.cx/p0f3/, accessed on 26 March 2021)—is a tool mainly known for its capabilities of OS fingerprinting. In version 3 of the tool, additional functionality of the HTTP fingerprinting was added. It can fingerprint client requests but also server responses. However, the request support is limited only to *GET* and *HEAD* methods, which in the authors' opinion is a huge drawback. The request fingerprint provides information about protocol version, order-preserving list of headers present in the request, absent headers, and *User-Agent* header value. When creating a list of headers, headers defined as optional are marked with a question mark "?", values of Host and User-Agent headers are skipped. For other headers, their values are provided, creating a list of header name and value pairs. If any of the headers *User-Agent*, *Host*, *Connection*, *Accept*, *Accept-Encoding*, *Accept-Language*, *Accept-Charset* or *Keep-Alive* is absent, such information is provided by the fingerprinter. Note that Hfinger, proposed in this paper, also provides an order-preserving list of headers, but header values are provided in a separate part of the fingerprint and only for popular headers.

p0f provides information about automatic fingerprint generation. The tool can also handle user-provided fingerprints and search in pcap files for such fingerprints. The search can identify requests even when some other headers occur between those present in the fingerprint or when some of the headers are missing—those that are marked as optional, providing that p0f can detect mismatches between the identified fingerprint and the declared *User-Agent* value. The Hfinger does not offer this functionality.

2.2.3. Mercury

Mercury is a network metadata capture and analysis framework (https://github.com/cisco/mercury, accessed on 26 March 2021). It provides fingerprinting capabilities for protocols such as TLS, DTLS, SSH, HTTP, and TCP. Additionally, it can perform application identification using the created fingerprints. The HTTP fingerprinting can be performed on both requests and responses. Note that the description presented below covers only request fingerprinting.

The tool analyzes the HTTP request to extract information about request method, protocol version, and a list of present headers, if they are on a predefined list of popular headers, including *Accept-Charset*, *Accept-Language*, *Cache-Control*, *Host*, and *User-Agent*. Some of the headers are presented with their values, for example *Accept*, *Accept-Encoding*, or *Connection*. All these features are represented using their hexadecimal values, forming the actual fingerprint. Beside the fingerprint, the tool provides contextual information that presents in a clear form URI and values of headers such as: *User-Agent*, *Host*, *X-Forwarded-For*, and *Via*.

2.2.4. Limitations of Current Tools

The described tools use a limited set of features for HTTP request fingerprinting, and the performed analysis is limited. FATT neglects URI, method, protocol version, payload, and headers' values during the fingerprint generation process. p0f does not analyze URI, payload, and method of request. Mercury does not process the payload, and URI analysis is limited to simple encoding that is even not added to the fingerprint. In both p0f and Mercury, the list of popular headers included in fingerprint creation can be improved as well as the list of headers whose values are added to the fingerprint.

The described tools' analysis is insufficient to achieve a satisfactory level of malware HTTP request fingerprinting uniqueness and can be improved. This paper will try to prove this statement by comparing the results of these three tools with the proposed approach. Moreover, to the authors' best knowledge, there is no extensive academic study that systematically analyzes the effectiveness of FATT, p0f, or Mercury for malware HTTP fingerprinting.

3. Hfinger

In this section, Hfinger functioning is presented in detail, along with request features that the tool investigates. Moreover, the process of the fingerprint generation is thoroughly explained.

3.1. Hfinger's Workflow

Hfinger has been created using Python3 language and additionally it utilizes TShark (https://www.wireshark.org, accessed on 26 March 2021, minimum version 2.2.0) to reassemble TCP segments into HTTP requests. An overview of the tool's workflow is presented in Figure 1.

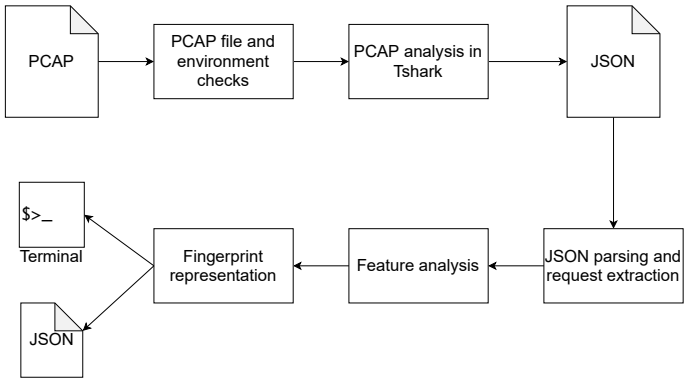


Figure 1. Hfinger's data workflow.

The tool firstly checks the execution environment to determine whether minimal criteria for running are met (e.g., Tshark binary is present) and if the input file is a valid PCAP file. If successful, the tool calls Tshark binary and feeds the analyzed PCAP file into it. TShark is configured to output a JSON file covering only HTTP requests. Then, the output JSON file is parsed by Hfinger and the requests are extracted. In the next step, the extracted requests are analyzed to generate the feature values. In the final step the feature values are joined together with a "|" (pipe) in a particular order, forming the HTTP request fingerprint. Depending on the users' choice, the results in the JSON format are either printed to the terminal or written to a file. The JSON output consists of the basic network information about each request: request timestamp, IP addresses, utilized ports, and the actual request fingerprint. Features analyzed by Hfinger are described in detail in the next section.

3.2. Analyzed Features

In this section, the features analyzed by Hfinger are presented and discussed. The chosen feature set utilized by the developed tool relies on the authors' previous work [17], previously published research (see [18] for URI features), and the authors' own malware analysis experience. In general, extracted features can be divided into three groups depending on the part of the request that they refer to: URI, headers, and payload.

3.2.1. URI Features

These features are used to extract information from the URI part of a request. They include:

- *Length of the URI*, represented as a logarithm with base 10 of the actual URI length (provided as a floating-point number rounded to one decimal place or rounded to an integer);
- *Number of directory levels in the URI*, represented as an integer;

- *Average length of the directory*, represented as a logarithm with base 10 of the actual average length of the directory (provided as a floating-point number rounded to one decimal place or rounded to an integer);
- *Extension of the file requested in the URI*, if applicable. The extension is extracted only if it is present on a defined list of popular extensions to prevent extracting nonsensical values;
- *Length of the variable part of the URI*, where the URI parameters are stored, represented as a logarithm with base 10 of the length (provided as a floating-point number to rounded one decimal place or rounded to an integer);
- *Number of variables in the URI*, represented as an integer;
- *Average value length*, represented as a logarithm with base 10 of the actual average value length (provided as a floating-point number rounded to one decimal place or rounded to an integer).

3.2.2. Header Structure Features

They provide information about headers, their values, extended with information about the request method, and HTTP version. The analyzed features consist of (in the order used in the fingerprint):

- *Request method*, presented as the first two characters of the method name;
- *HTTP version*, expressed as a single number, depending on the first digit after the dot in the protocol definition, for example, "1" for "1.1" version and "9" if no protocol version is defined;
- *Representation of header order in the analyzed request*, where the headers are expressed by the chosen encoding scheme. This scheme provides a list of popular headers for which encoding is provided to shorten the fingerprint length. However, if the header is not on the list, its name is hashed using the 32-bit Fowler–Noll–Vo hash function in version 1a (FNV1a) [19], and the hexadecimal representation of the hash is used as the name. If the header name does not begin with an upper case letter (or any first letter of the parts of a compound header name, for example, *Accept-Encoding*), an exclamation mark *!* is prepended to the header representation;
- *Representation of popular header's values*—the following headers are analyzed to extract their value:
  - *Connection*,
  - *Accept-Encoding*,
  - *Content-Encoding*,
  - *Cache-Control*,
  - *TE*,
  - *Accept-Charset*,
  - *Content-Type*,
  - *Accept*,
  - *Accept-Language*,
  - *User-Agent*.

  If the value is present on a list of popular values, it is encoded with a chosen short encoding representation. If it is not on the list, the values are hashed using FNV1a. The representation is provided as an encoded header name and its encoded value, separated by ":" (colon), and such pairs are separated using "/" (forward slash). If the header can have multiple values, their representation is separated by "," (comma). The order of the headers is preserved. Additionally, the value of the *User-Agent* header is always represented as the FNV1a hash.

3.2.3. Payload Features

They are extracted if the payload of a request is not empty. Payload features consist of three features (in the order used in the fingerprint):

- *Presence of non-ASCII characters*, represented as a single letter "N" if non-ASCII characters are present, and "A" otherwise,
- *Shannon entropy of the payload*, represented as a floating-point number rounded to one decimal place or rounded to an integer,
- *Payload length*, represented as a logarithm with base 10 of the actual payload length (provided as a floating-point number rounded to one decimal place or rounded to an integer).

3.2.4. Numerical Features' Representation

As presented above, some of the numerical features are inherently real numbers and have to be represented as a float type. During the design phase, a decision was made to round such values to one decimal place or round them to an integer. The rounding mode can have a significant impact on Hfinger evaluation; thus, in Section 5.1 an analysis of which version of the representation should be chosen for each of these features is discussed.

3.3. Fingerprint Generation

Features described above are used to create the HTTP request fingerprint. Figure 2 illustrates an overview of an exemplary Hfinger fingerprint generation. All analyzed features are presented, including floating-point representation, what may vary from the final feature set selection presented in Section 5.1.

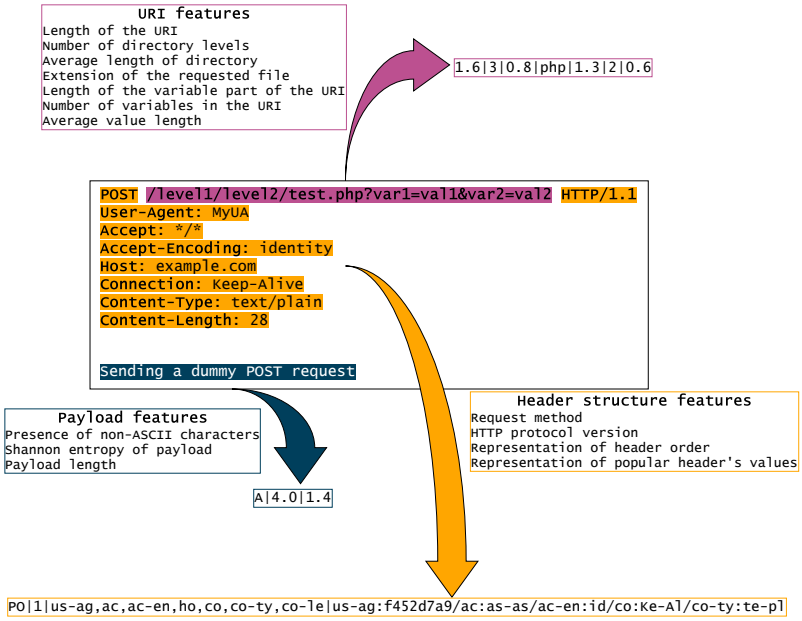


Figure 2. An example of a HTTP POST request fingerprint produced by Hfinger.

As presented in Figure 2, Hfinger analyzes three parts of the HTTP request to generate a fingerprint. Firstly, the URI part is analyzed and the feature values are generated. For example, in Figure 2, the URI length is 43 characters, there are 3 directory levels, and a *PHP* file is requested. These features are represented in the generated fingerprint part as *1.6*, *3*, and *php* respectively.

Secondly, header structure features are extracted to generate the second part of the final fingerprint. For instance, using the example in Figure 2, the method is *POST*, protocol is in version *1.1*, *User-Agent* header has value of *MyUA*, and *Connection* header has value of *Keep-Alive*. These values are transformed into corresponding fingerprint parts: *PO*, *1*, *us-ag:f452d7a9*, and *co:Ke-Al*, respectively. Header names on their own and in pair with values are parts of broader structure features, representing order of all headers in the

request, or representing popular header's values, also order wise. The encoding of header names and values is provided by Hfinger's configuration file.

The third part of the fingerprint is generated on the basis of request's payload data. In the example in Figure 2, the request contains payload of *Sending a dummy POST request*. This string is built only from ASCII characters and is 28 characters long. The corresponding features generated by Hfinger are *A* and *1.4*.

Final fingerprint is created by combining the three generated parts in predefined manner: URI features, header structure features, and payload features. The fingerprint length is variable as it is dependent on the request's structure and data, for example, payload features are provided only if payload data is present. The final selection of features in particular fingerprint parts and their rounding mode also affects the format and length of the fingerprint. This will be described in Section 5.1.

4. Experimental Methodology

In this section, we first present details related to the malware and benign application data sets that are later used during experimental evaluation. Next, we describe existing fingerprinting tools utilized while conducting comparison analysis. Finally, we outline and define performance measures and experimental methodology.

4.1. Malware Data Set

Malware data set is compiled from two pcap data sets: one was used in the authors' previous work [17] and originates from CERT Polska's sandboxing environment and Malware Capture Facility Project (https://www.stratosphereips.org/datasets-malware, accessed on 26 March 2021). The second data set was derived from a newer version of CERT Polska's malware analysis platform.

The first data set contains 26,133 pcap files analyzed and labeled in the previous work. To this end, Snort IDS with Emerging Threats Pro (ET Pro—https://www.proofpoint.com/us/threat-insight/et-pro-ruleset, accessed on 26 March 2021) and Snort Registered (https://www.snort.org/downloads#rules, accessed on 26 March 2021) rulesets were used. More information about this data set can be found in [17].

The second pcap data set consists of 8674 files and it was created specifically for the purpose of this research. The analyzed pcap files originate from the CERT Polska's malware analysis platform, where Windows-based malware samples are analyzed. The malware samples are obtained from various open-source feeds, for example, Abuse.ch (https://abuse.ch/, accessed on 26 March 2021), from external user uploads via mwdb.cert.pl, accessed on 26 March 2021 malware service, and from the CERT Polska's internal malware hunting systems. The analyzed pcap files were labeled using Suricata IDS and ET Pro ruleset. The labeling process was performed in multiple steps. Firstly, all pcap files were analyzed using Suricata IDS and these logs were saved. Secondly, alert messages from the IDS logs were reviewed semimanually to include only those related to HTTP requests and the malware family's name. Based on the SID rule identification number, the alert messages were labeled with the corresponding malware family name using the information from the corresponding IDS rule. Thirdly, all HTTP requests belonging to a particular network flow, for which the reviewed IDS alert existed, were labeled with corresponding alerts. This step was performed with the assumption that all requests within such a network flow should be treated as malicious. Network flows were identified by source and destination IP addresses and ports. Note that in many cases, HTTP requests were labeled with multiple IDS alerts. As the last step, malware requests were labeled with the malware family name. Again this process was performed semiautomatically by reviewing the names of families corresponding to alerts of particular requests. Requests with multiple different family names were analyzed manually. In most cases, it involved aliases of malware, when names were merged to one, or forks of malware families. These were merged to one name or a specific fork name was chosen. For example, all Ursnif

family forks were merged, because alert messages were written by different rule authors, thus incorporating inconsistencies in naming, and the provided fingerprints were identical.

The two data sets mentioned above were merged based on the labeled malware family name. They originate nearly from the same source of malware traffic and use the same intelligence source for labeling, mainly ET Pro rulesets. The final malware data set used in further analyses covers 121 popular malware families with 401,566 HTTP requests. The complete data set provides more data; however, only those malware families were chosen that have at least 20 requests. The top 10 malware families sorted by the number of HTTP requests are presented in Table 2, while the complete list is presented in Appendix C.

Table 2. Top 10 malware families by the number of HTTP requests in the final data set.

Malware Family Name	Number of Requests	Percentage of All Requests [%]
Upatre	62,257	15.50
Simda	57,730	14.38
Locky	44,498	11.08
Dridex	30,070	7.49
Arkei	22,057	5.49
DirtJumper	18,486	4.60
Chthonic	14,410	3.59
Vflooder	14,252	3.55
Ursnif	11,756	2.93
Arid Viper APT	10,063	2.51

4.2. Data Set of Benign Application

Apart from the malicious data set, the benign one was also necessary. To obtain it, network traffic of benign applications was collected from two sources: (i) popular web browsers, including background traffic, and (ii) popular benign applications running on Windows 10.

4.2.1. Popular Web Browsers

The data set of popular web browsers' network traffic was generated by the authors in their previous research [17], where it is described in detail. Various web browsers under the control of different versions of the Windows OS were used to visit websites from the list of 500 most popular websites worldwide, extracted from Alexa top 1 million websites worldwide (http://s3.amazonaws.com/alexa-static/top-1m.csv.zip, accessed on 9 February 2017). The websites were accessed between 9 and 15 February 2017 and between 13 and 18 October 2017, depending on the browser. Table 3 contains information about the networking environment and the number of requests observed in each web browser traffic. Including background traffic, this part of the data set contains 194,940 HTTP requests.

Table 3. Networking environments in which web browser HTTP traffic was analyzed.

Browser Name	Operating System	Number of Requests
Microsoft Edge	Windows 10	17,659
Google Chrome	Windows 7	30,281
Mozilla Firefox (Adobe Flash Player installed)	Windows 7	19,523
Mozilla Firefox	Windows 7	26,131
Microsoft Internet Explorer 11	Windows 7	29,216
Google Chrome	Windows 8.1	22,133
Mozilla Firefox	Windows 8.1	19,082
Microsoft Internet Explorer 11	Windows 8.1	19,807

4.2.2. Network Traffic of Popular Benign Applications Running on Windows 10

Network traffic of popular benign applications running on Windows 10 was obtained using an experimental environment equipped to perform a man-in-the-middle (MitM) attack on HTTPS network traffic. The main objective was to create a data set of network traffic that would be highly similar to the traffic observed in a home or a small business network.

The experimental environment consisted of two virtualized hosts: one, in the remainder of this section called analysis host, was running Windows 10, and the second one, called *MitM host*, was used to provide Internet connectivity and network traffic dumping. Windows 10 OS was obtained from https://modern.ie, accessed on 26 March 2021 in version 1809. Additional root X.509 certificate was installed in the system to provide means for the MitM mechanism. *MitM host* was based on Ubuntu 20.04 LTS OS, equipped with sslsplit tool to perform a man-in-the-middle attack on HTTPS traffic. All network traffic was routed through the *MitM host*. The experiment was divided into three parts that were executed during six consecutive days. The network traffic was not deciphered during the first part, mainly giving unmangled situation and normal traffic. All OS updates were performed during this period.

In the second part of the experiment, the MitM mechanism was enabled and network traffic on ports 80 and 443 was forwarded through sslsplit. sslsplit was configured to work with the least offensive mode to minimize its impact on the network traffic. The traffic was dumped to pcap files for later analysis. During this period, popular benign applications were installed and run. This includes VLC media player, Adobe Acrobat Reader, Steam, Spotify, Discord, Libre Office, and Microsoft Office. The complete list is available in Appendix B. The applications were used to mimic the behavior of a standard user: creating files with Microsoft Office/LibreOffice suites, saving them to OneDrive cloud repositories, opening some saved files, using e-mail clients to download and send messages, listening to music, or downloading files. In all applications, update modules were used to download any available updates. Additionally, some well-known websites were visited using Google Chrome and Microsoft Edge based on Chromium, including registering and logging on popular social media sites such as Facebook, Instagram, and Outlook.com, accessed on 26 March 2021. Internal Windows applications were also used, including weather, calendar, and movie services. Usage of the MitM during this part of the experiment caused some essential OS services to stop working, including Windows Update and Windows App Store. According to multiple sources [20,21], these services send telemetry data using HTTPS with internal, additional X.509 certificate repository and certificate pinning mechanism. When sslsplit was enabled, these applications encountered the error 80245006.

In the third part, the sslsplit was disabled and traffic was dumped in the same manner as during the first part. This phase provided an environment not impacted by the MitM mechanism, with all OS services operating normally and background services of previously installed applications.

The data set contains 53,717 HTTP requests. The top 10 values of the *User-Agent* header value ordered by the number of requests are presented in Table 4. Note that 2.26% of requests do not contain *User-Agent* header or its value is empty.

Table 4. The top 10 values of *User-Agent* header value ordered by the number of requests in the data set of network traffic of popular benign applications running on Windows 10.

***User-Agent* Header Value**	**Percentage of All Requests in the Data Set [%]**
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.116 Safari/537.36 Edg/83.0.478.58	35.87
Microsoft-Delivery-Optimization/10.0	10.57
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.116 Safari/537.36 Edg/83.0.478.61	9.46

Table 4. *Cont.*

***User-Agent* Header Value**	**Percentage of All Requests in the Data Set [%]**
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.116 Safari/537.36	6.50
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.140 Safari/537.36 Edge/18.17763	5.27
Mozilla/5.0 (Windows; U; Windows NT 10.0; en-US; Valve Steam Client/default/1591251555;) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.117 Safari/537.36	2.82
Mozilla/5.0 (Windows NT 10.0.17763; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Slack/4.7.0 Chrome/83.0.4103.119 Electron/9.0.5 Safari/537.36 Sonic Slack_SSB/4.7.0	2.52
Valve/Steam HTTP Client 1.0 (0)	1.65
microsoft.windowscommunicationsapps	1.51
Microsoft Office/16.0 (Windows NT 10.0; Microsoft Outlook 16.0.13001; Pro)	1.45

4.3. Fingerprinting Tools Used for Comparison

Three HTTP fingerprinting tools were used for comparison with Hfinger: FATT, p0f, and Mercury. Their overview is presented in Section 2.2. Their source code versions are presented in Appendix A. Code changes and configuration of the tools used in the analysis are described below.

FATT is used in the version provided by its GitHub repository (https://github.com/0x4D31/fatt/, accessed on 26 March 2021). Additionally, to provide similar test conditions between all tested tools, two types of FATT output are further analyzed. The first one is the header hash as provided by the tool. The second one is the header hash with the value of the *User-Agent* header that is the output of the default reporting mode when used with the command-line interface.

p0f was analyzed using source code parts of its Python port (https://github.com/FlUxIuS/p0f3plus, accessed on 26 March 2021). As the tool fingerprints only *GET* requests, the code was patched to analyze all request types to provide the same base for comparison with Hfinger. The code was also patched to support requests with a nonstandard end of line tag: *LF* instead of *CRLF*.

For analysis of Mercury its Python version *pmercury* was used. Mercury's analysis process can be modified using a configuration file to manipulate, for example, the list of analyzed headers or the list of headers that should be represented with their values. Thus, for comparison with other tools, two configurations were used: (i) the default, provided by the authors of the tool and (ii) the same as the default but extended with representing the value of the *User-Agent* header in the fingerprint. The source code was patched to support the analysis of nonstandard requests. These were present when analyzing requests with the nonstandard end of line tag: *LF* instead of *CRLF* and those without protocol version definition. Even though such requests are rarely observed in malware traffic, they should be properly handled.

4.4. Comparison Measures and Methodology

In this research, the performed analyses and comparisons are based on four measures that, in our opinion, provide useful insights into real-life applications of malware HTTP traffic fingerprinting tools. This includes, for example, the uniqueness of the fingerprint across malware families.

Please note that to minimize the effect of different sizes of request sets of analyzed malware families, the measures are computed as averages of each family's partial value, not a global value. Firstly, the analyzed phenomena occurrences are counted separately for each family and then the average value is computed and provided as the final measure. With such an approach, all requests in each malware family set can be analyzed, which

could not be achieved if, for example, undersampling methods of data set balancing were used. If the measures were counted with a global approach, the families with a significantly higher number of requests (e.g., Dridex, Upatre, or Chthonic) would bias measure's value impacting the whole analysis.

The comparison measures utilized in this research include: malware collision level, fingerprint generation level, level of collision with benign applications, and entropy. All of them are explained in detail below.

Malware collision level provides information on whether any collisions of request fingerprints between malware families occur, that is, whether request fingerprints are seen across multiple families and are not unique to one family. This measure should be as low as possible to provide exclusive and reliable fingerprints. Malware collision level is computed by firstly counting the ratio of requests with fingerprint collision to all requests for each family, then counting the mean value of these ratios across all families. Malware collision level is expressed by Equation (1), where N is the number of malware families, r_i^c denotes the number of requests with fingerprint collision for malware family i, and r_i expresses the number of all requests for malware family i.

$$C_M = \frac{\sum_{i=1}^{N} \frac{r_i^c}{r_i}}{N} \tag{1}$$

Fingerprint generation level provides information about the number of fingerprints generated for a particular malware family set of requests. It can be interpreted as a measure of an average number of fingerprints generated per analyzed request set (for example, in a single pcap file) but also a measure of the degree to which requests are grouped together. Thus indirectly informing about the degree of a fingerprinting tool's request information generalization. This measure should be as low as possible but still capable of discerning requests that are actually different. It results from requirements that fingerprinter should extract only necessary information from requests and minimize the number of produced fingerprints, not to overwhelm logging and analytic systems. The measure is computed by counting the average ratio of request fingerprints to all malware families' requests. Fingerprint generation level is calculated using Equation (2), where N is the number of malware families, f_i is the number of fingerprints for malware family i, and r_i is the number of all requests for a malware family i.

$$G = \frac{\sum_{i=1}^{N} \frac{f_i}{r_i}}{N} \tag{2}$$

Note that a trade-off between malware collision and fingerprint generation levels exists. When the fingerprinter extracts more information from requests to provide a more unique set of fingerprints, it decreases the collision level. However, it also provides a larger number of fingerprints, as a result increasing the fingerprint generation level. This trade-off is further analyzed in Section 5.1, where optimization of these measures is performed.

The third measure is the level of *collision with benign applications* that provides information about the number of malware fingerprint collisions with some popular, benign applications. In real-life environments, malware operates along with standard, nonmalicious applications. A good fingerprinter should be capable of producing unique fingerprints both to malware and benign applications, thus providing means for discerning these types of applications. This measure is computed similarly to *malware collision level*, i.e., an average value of the ratio between malware requests with fingerprint collision with benign applications and the number of all requests. The level of collision with benign applications is expressed by Equation (3), where N is the number of malware families, r_i^{bc} expresses the

number of requests with fingerprint collision with benign applications for malware family i, and r_i denotes the number of all requests for the malware family i.

$$C_B = \frac{\sum_{i=1}^{N} \frac{r_i^{bc}}{r_i}}{N} \tag{3}$$

The final, fourth measure provides information about *entropy* of the tool. It is represented as an average Shannon entropy of fingerprints across analyzed malware families represented in bits. Equation (4) provides the formula of this measure, where N is the number of malware families and H_i Shannon entropy for malware family i.

$$E = \frac{\sum_{i=1}^{N} H_i}{N} \tag{4}$$

Shannon entropy H_i of fingerprints of a particular malware family i is defined by Equation (5), where M_i denotes the number of fingerprints produced by the tool for the malware family i, p(f_j) represents the occurrence probability of fingerprint f_j (computed as the number of occurrences of requests with a particular fingerprint j divided by the number of all requests of the particular family), and log_2 is a logarithm with base 2.

$$H_i = \sum_{j=1}^{M_i} p(f_j) log_2(p(f_j)) \tag{5}$$

Fingerprint entropy E can be interpreted as a measure of the average amount of information provided by malware fingerprints of a particular tool. The higher the value, the better, as in this case, fingerprints are more informative.

5. Experimental Results

Below we present the experimental evaluation of the proposed Hfinger tool. First of all, we demonstrate how the optimal feature set selection has been performed. Then, we outline the results of the comparison of Hfinger with other existing HTTP fingerprinting tools. Note that all tools were analyzed using fingerprint exact match search, and no fuzzy search mechanisms were utilized, even if the tool under evaluation supports it.

Data sets presented in Section 4 were divided randomly into two equal parts based on the malware family (malware data set) or the application name present in the *User-Agent* string (benign data set). For each malware family/application, 50% of the requests were assigned to the first part used to select the optimal feature set, while the rest of the requests were assigned to the part used for the final evaluation of fingerprinting tools.

5.1. Selecting the Optimal Feature Set

The goal of the selection of the optimal feature set is to provide a list of features from those presented in Section 3.2 that will provide the optimal results of the four measures defined in Section 4.4 (i.e., malware collision level, fingerprint generation level, level of collisions with benign applications, and fingerprint entropy). Additionally, some numerical features can be presented with different rounding: with or without fractional component, in the remainder of the text described as a float or as an integer, respectively. Thus, this process will provide information on which rounding would be best for each feature.

The process of feature set selection is based on two steps. Firstly, the defined measures are computed for all 186,623 subsets of features. Secondly, the actual selection was performed using different methods described further in the text, including multiobjective optimization techniques, with the results from the first step.

Figure 3 presents the relationships between all pairs of defined measures: malware collision level, fingerprint generation level, level of collisions with benign applications, and fingerprint entropy for all possible combinations of feature sets.

Analysis of the relationship diagrams in Figure 3 suggests that for many feature sets, with the increase of malware collision levels, the level of collisions with benign applications also increases, while the fingerprint generation level and fingerprint entropy decrease. This follows the intuition that with the increase of fingerprint information, fewer malware requests are incorrectly tagged with the exact representation but for the price of an elevated number of fingerprints.

Based on Figure 3 an interesting observation can also be made. Two distinct result groups can be seen for all diagrams except for fingerprint generation level and fingerprint entropy relationship. One of the groups represents results better suited for optimization. Analysis of feature sets showed that this group contains sets with the order of headers or popular headers and their values, thus indicating a significant impact of those two features on results.

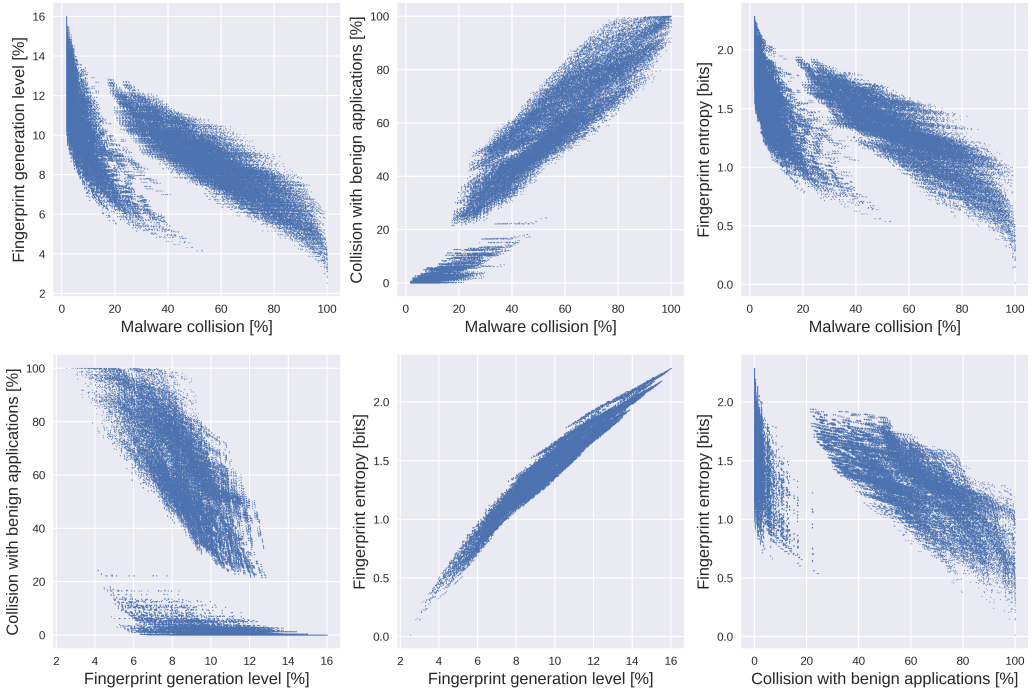


Figure 3. Relationships between defined measures for all possible combinations of feature subsets using training data set. From left to right upper row: (**a**) fingerprint generation level in function of malware collision level, (**b**) level of collision with benign applications in function of malware collision level, (**c**) fingerprint entropy in function of malware collision level. From left to right lower row: (**d**) Level of collision with benign applications in function of fingerprint generation level, (**e**) Fingerprint entropy in function of fingerprint generation level, (**f**) Fingerprint entropy in function of collision level with benign applications.

Using results described above, five feature sets (A–E) are selected, and they are presented in Table 5.

The feature sets (A–E) were selected using the following methods. The descriptions include a short explanation of the main motive behind each selection method:

A lexicographic method from multiobjective optimization techniques (see [22]), where firstly malware collision level was minimized, then using this minimal value, the minimal value of fingerprint generation level was selected. Obtained feature sets had equal values of collision level with benign applications and fingerprint entropy. Finally, a feature set with the lowest number of features was chosen. This set has been chosen using a proven method of multiobjective optimization.

B from all feature sets with maximal feature number, the set with the minimal level of fingerprint generation level was selected. One such set existed. This set has been chosen to provide information about all proposed request features but also to minimize number of generated fingerprints. In some analysis scenarios, e.g., highly similar malware, such complete information might be crucial for discerning malware families.

C firstly, feature sets were limited to those containing features from the defined list. The list was compiled based on the authors' experience with malware analysis and how commonly such features are used in their operational work. The list is formed by features: length of the URI, the extension of the file requested in the URI, representation of header order or representation of popular header's values (at least one feature from this pair), request method, protocol version, payload length, and Shannon entropy of payload. Secondly, the feature sets were filtered to provide only those with a null level of collisions with benign applications and a minimal level of collisions with malware. The feature set with a higher number of features was chosen from two sets with an identical value of fingerprint generation level and fingerprint entropy.

D firstly, fingerprint generation level was limited to 6% (approximately half of the value observed for the sets chosen with the lexicographic method). Then, feature sets with the lowest number of malware collisions were chosen. From four such sets, one with the lowest number of features was chosen. This set has been chosen to provide significantly lower fingerprint generation levels than other Hfinger's feature sets that also are comparable to other tools.

E firstly feature sets with the maximum level of fingerprint entropy were chosen, and then feature sets with minimal fingerprint generation level were chosen. Four feature sets were obtained with this method, where malware collision level and level of collision with benign applications were equal. Hence, the set with the highest number of features was chosen. This set has been chosen to provide the highest entropy level but with the minimal possible number of generated fingerprints, thus giving the most informative fingerprints from all feature sets.

Table 5. Selected feature sets.

Feature Set Name	Feature List
A	average directory length represented as an integer average value length represented as a float number of directories extension of requested file order of headers popular headers and their values payload length represented as a float
B	average directory length represented as an integer average value length represented as an integer number of directories extension of requested file URI length represented as an integer variable length represented as an integer number of variables request method version of protocol order of headers popular headers and their values presence of non-ASCII characters payload entropy represented as an integer payload length represented as an integer

Table 5. *Cont.*

Feature Set Name	Feature List
C	average directory length represented as an integer average value length represented as a float number of directories extension of requested file URI length represented as an integer request method version of protocol order of headers popular headers and their values presence of non-ASCII characters payload entropy represented as an integer payload length represented as a float
D	average directory length represented as an integer average value length represented as an integer extension of requested file URI length represented as an integer order of headers
E	average directory length represented as a float average value length represented as a float number of directories extension of requested file URI length represented as a float variable length represented as a float request method version of protocol order of headers popular headers and their values presence of non-ASCII characters payload entropy represented as a float payload length represented as a float

The results for these five feature sets, along with the results for other analyzed, existing tools, are presented in Table 6.

Table 6. Optimization results for five selected feature sets compared to other analyzed tools. The *UA* suffix marks nondefault configuration of tools supporting *User-Agent* header value as a part of the fingerprint.

Tool	Malware Collision Level [%]	Fingerprint Generation Level [%]	Level of Collisions with Benign Applications [%]	Fingerprint Entropy [bits]
Hfinger (A)	1.76	11.76	0.00	1.72
Hfinger (B)	3.49	11.19	0.00	1.57
Hfinger (C)	1.76	12.09	0.00	1.77
Hfinger (D)	16.85	5.95	1.11	0.87
Hfinger (E)	1.76	15.96	0.00	2.29
FATT	53.04	4.16	24.45	0.54
FATT UA	22.11	6.63	11.87	0.88
Mercury	64.15	4.11	31.33	0.49
Mercury UA	27.13	6.58	15.26	0.85
p0f	15.70	16.71	11.25	1.99

5.2. Comparison of Hfinger to Other Existing Tools

A final comparison of the results was performed using the remaining 50% of the data set, as described in Section 5. The results for the four defined measures are presented in Table 7.

Based on the results presented in Table 7, a general observation can be made that with the default configuration, Mercury provides the worst levels of collision (both malware and benign applications) and fingerprint entropy for all analyzed tools, 63.34%, 31.95%, and 0.46 bits, respectively. It is followed by FATT with the default configuration with malware collision level at 53.45%, benign application collision level at 25.11%, and fingerprint entropy at 0.51 bits. Nevertheless, these two tools provide the lowest fingerprint generation levels: 3.79% for Mercury and 3.83% for FATT. p0f compared to these two tools results in a lower level of malware and benign applications collisions (15.25% and 10.96% respectively) and a higher level of fingerprint entropy (1.98 bits) but at the cost of a higher fingerprint generation level, i.e., 16.41%.

Table 7. Final evaluation of Hfinger's five selected feature sets compared to other analyzed tools. The *UA* suffix marks nondefault configuration of tools supporting *User-Agent* header value as a part of fingerprint.

Tool	Malware Collision Level [%]	Fingerprint Generation Level [%]	Level of Collisions with Benign Applications [%]	Fingerprint Entropy [bits]
Hfinger (A)	1.85	11.76	0.00	1.72
Hfinger (B)	3.58	11.01	0.00	1.58
Hfinger (C)	1.85	12.15	0.00	1.78
Hfinger (D)	16.78	5.78	1.51	0.85
Hfinger (E)	1.78	15.96	0.00	2.30
FATT	53.45	3.83	25.11	0.51
FATT UA	21.77	6.32	12.22	0.87
Mercury	63.34	3.79	31.95	0.46
Mercury UA	26.46	6.27	15.76	0.84
p0f	15.25	16.41	10.96	1.98

When the *User-Agent* header value is used as a part of a fingerprint for FATT or Mercury, the tools provide lower levels of collisions, both for malware and benign applications but also higher fingerprint entropy. For FATT, the malware collision level decreases by nearly 32 pp (percentage points), from 53.45% to 21.77%, while for Mercury, it decreases by almost 37 pp, from 63.34% to 26.46%. Collisions with benign applications decrease by nearly 13 pp from 25.11% to 12.22% for FATT and by 16 pp from 31.95% to 15.76% for Mercury. An increase in fingerprint entropy value is observed from 0.51 to 0.87 bits for FATT and from 0.46 to 0.84 bits for Mercury. These improvements of the three measures' values come with the worsening of the fingerprint generation level that nearly doubles both for FATT (from 3.83% to 6.32%) and Mercury (from 3.79% to 6.27%). These results support intuition of the relationship between the collision level and the fingerprint generation level, i.e., if the tool better discerns applications, the number of fingerprints it provides also elevates.

Further analysis of Table 7 shows that, except feature set D, all other feature sets of Hfinger provide significantly lower levels of malware collision than other tools. Feature sets A, C, and E achieve levels lower than 2% and feature set B lower than 4%. These levels are lower by nearly 60 pp (30 times), compared to the worst value for Mercury, and at least 11 pp (4 times) when compared to the best value of the p0f tool. These feature sets also provide a null value of collisions with benign applications that is lower by 10 to 30 pp when compared to other tools. Hfinger's feature sets A, B, and C achieve higher fingerprint entropy levels than Mercury and FATT, i.e., 1.72, 1.58, and 1.78 bits, respectively. It is about one bit higher than these two tools. However, only feature set E that was chosen to provide the maximum fingerprint entropy of 2.29 bits achieves a higher level than p0f. As observed before with other tools, Hfinger's measure improvements increase fingerprint generation level. For feature sets A, B, and C, they are nearly 12% (11.76%, 11.01%, and 12.15% respectively), and for feature set E, it is 15.85%. These values are higher by 5 to 12 pp (2–3 times) compared to FATT and Mercury but lower by 5 pp or nearly equal compared to p0f.

Hfinger's feature set D was chosen with a focus on providing a lower fingerprint generation level than other feature sets, more comparable to FATT and Mercury; thus, it will be analyzed separately for the sake of brevity of the argument. As Table 7 presents, feature set D achieves lower collision levels for both malware and benign applications when compared to FATT and Mercury. Differences appear in comparing feature set D with the default and nondefault configurations of these two tools. Default configurations achieve lower fingerprint generation levels by nearly 2 pp compared to feature set D but almost the same level for nondefault configurations. The same is observed for fingerprint entropy: default versions achieve lower levels, while nondefault achieve nearly the same values as Hfinger with feature set D. When compared to p0f, feature set D achieves 1.5 pp higher malware collision level, but 10 pp lower fingerprint generation level. The level of collisions with benign software is also lower for feature set D by 9 pp. However, p0f achieves a higher fingerprint entropy level: 1.98 bits when the feature set D: 0.85 bits. The additional perspective of the results for feature set D is provided by the fact that this feature set does not contain information about the values of *User-Agent* header, unlike other Hfinger's feature sets and unlike p0f, and FATT's and Mercury's nondefault configurations. It can be used as a starting point for future work on analyzing how much the *User-Agent* header's value can impact the fingerprint and its capabilities to identify applications.

Overall, the analysis results show that in the majority, Hfinger achieves significantly lower levels of malware and benign applications collisions than other analyzed tools. It results in higher fingerprint generation levels compared to FATT and Mercury but still lower than those of p0f. Fingerprint entropy for Hfinger is also higher than that of FATT and Mercury; however, only one feature set achieves a higher level of this measure than p0f. Specifically designed to decrease the fingerprint generation level, feature set D achieves lower levels of collisions with malware and benign applications when compared to FATT and Mercury but with similar or only 2 pp higher fingerprint generation level. It also provides similar or higher fingerprint entropy. This feature set produces 1.5 pp more malware collisions than p0f and achieves lower fingerprint entropy and has lower levels of fingerprint generation and collisions with benign applications.

Regarding the above analysis, feature set C has been chosen as a default reporting mode for the Hfinger. Firstly, it provides a similar malware collision level as sets A and E, lower than sets B and D. Secondly, its fingerprint generation level is similar to that of set A and lower than of set E. Thirdly, its fingerprint entropy level is lower than of set E but almost identical of set A. Lastly, feature set C provides information about a higher number of features than set A, giving a more complex overview of a request for analyst. Feature set E achieves this with a higher fingerprint generation level.

6. Practical Usage Scenarios and Limitations

In this section practical usage scenarios for Hfinger are presented along with the discussion on limitations of this tool.

6.1. Practical Usage Scenarios

Hfinger was designed to be used as a standard network fingerprinting tool, and its usage cases are no different from other tools. It is capable of reading pcap files, thus it can analyze network traffic originating from different sources, for example, malware sandbox systems, honeypots but also enterprise networks. Hfinger can be used directly to analyze network traffic or it can be used as a subsystem, whose output is ingested by other analysis systems. When used as a standalone tool, it can help the analyst in network forensic objectives. While used as a subsystem, it can feed data into network monitoring or event logging systems, for example, SIEM (Security Information and Event Management) solutions.

Fingerprints created by Hfinger can also be used to identify and track malware in different scenarios. For example, when analyzing the network traffic of unknown malware, fingerprints created by Hfinger can be used to identify requests that were previously

labeled as belonging to a particular malware family. Moreover, if the analyzed network traffic consists of multiple HTTP requests, fingerprints can help in grouping them, giving a basis for further analysis of the purpose of the requests—whether it was a connectivity check, C&C server check-in, or some other malicious activity. Additionally, Hfinger can extend and complement IDS systems by using alerts to search for requests that were not reported but have the same fingerprint as those identified. Another application is to identify similarities between different malware families when similar fingerprints for both are discovered; however, this can be achieved when using Hfinger with techniques other than default exact match search. These were not analyzed in this paper but can be used as a starting point for future work.

Hfinger cannot analyze HTTP network traffic secured with HTTPS protocol on its own. However, in many environments and architectures, HTTPS traffic can be inspected, for example, by using TLS keys in sandbox systems or TLS inspection systems in corporate networks.

6.2. Limitations

The presented research and the proposed tool, apart from the auspicious results as outlined in the previous sections, have their limitations that will be discussed below.

Firstly, the authors put maximum effort into the correct preparation of the data sets; however, not all biases could be eliminated. The malware labeling process involved the usage of the ET Pro IDS ruleset. It is a well-known, industry-tested intelligence source that both false positive and false negative errors could be present. That is, some benign HTTP requests were marked as malware, some malicious requests were not alerted, or the malware name provided by the rule was incorrect. Additionally, although malware families were carefully selected for the analysis, their distribution in terms of malware types might not reflect the actual distribution. These biases could influence the results of the analysis and Hfinger feature set selection process.

Secondly, Hfinger capabilities to analyze many features, including header values, can be less efficient for malware families that introduce many changes in the request structure. This can happen, for example, with malware used to perform DDoS attacks, where it is a common technique to change the value of *User-Agent* header with each request. In such a situation, the number of fingerprints created by Hfinger can increase. However, thanks to the fingerprint's modular structure, this issue can be addressed by ignoring the part of the fingerprint generating the higher level of noise.

Thirdly, some malware families tend to incorporate mimicking mechanisms to become similar to benign applications. Depending on the level of mimicry, Hfinger can help to uncover it. If the changes are simple, for example, altering the value of the *User-Agent* header to a benign one, the generated fingerprints will show only a change in this value. However, when the whole structure of a request is changed, then the issue is becoming harder to address. In the worst-case scenario, the request can be changed to such a degree that the malware fingerprint can be the same as of a benign application. Nevertheless, we believe that applying such a degree of mimicry mechanisms would require a lot of design effort and, from our experience, is not typical for malware developers.

Finally, fingerprints produced by Hfinger were designed to be used in exact match searches. Potentially, they can also be used to perform fuzzy searching, for example, by using only some parts of the basic fingerprint. However, we considered it is out of the scope of this paper. The main focus during Hfinger design was put on exact matching, which is supported by many security monitoring and logging tools, contrary to fuzzy search. Furthermore, fuzzy search functionality can be provided with different mechanisms, depending on the monitoring system, thus creating problems with interoperability and potential lack of support of some operations. These issues were analyzed during the design phase of Hfinger, and the decision was made to develop a solution that can be easily integrated into existing deployments of various systems and tools. Nevertheless, fuzzy searching or request clustering mechanisms can be treated as our future work.

7. Conclusions

This paper presents Hfinger, an HTTP request fingerprinting tool. Hfinger analyzes the network traffic and extracts information from different parts of the HTTP requests to provide a simple and interpretable for analyst representation of requests. The fingerprints provided by the tool can be used for exact match searches to identify similar requests between different pcap files and, as such, aid in threat hunting or as a step to identify unknown malware.

The results presented in this paper show that in the default Hfinger reporting mode, the generated fingerprints are 8–34 times more unique between malware families than in other three similar, community-proven, existing fingerprinting tools: FATT, Mercury, and p0f. In the default reporting mode, Hfinger introduces no collisions between malware and benign applications, contrary to the other tools. The number of generated fingerprints is at most about three times higher when compared to FATT and Mercury but 35% lower compared to p0f. Hfinger achieves higher levels of fingerprint entropy than FATT and Mercury but only slightly lower than p0f. In the authors' opinion, the three-fold increase in the number of fingerprints is justifiable by the significant (8–34 fold) increase of fingerprint uniqueness. Thus, this analysis confirms that Hfinger is an effective tool for malware HTTP request fingerprinting.

Hfinger can also operate in other reporting modes that can help achieve better fingerprint entropy levels, provide a lower number of fingerprints, or produce information about a broader set of request features. They offer better or at least comparable results for all measures defined in this paper compared to the other analyzed tools.

Future work will focus on enabling fuzzy searching. This includes, for example, capabilities for searching similar requests on the base of a fingerprint's substring, using a wildcard search or searching depending on the importance of fingerprint elements. Another direction is to use Hfinger as a basis for request clustering mechanisms, which can help to uncover new relations between requests.

Author Contributions: Conceptualization, P.B. and W.M.; Data curation, P.B.; Formal analysis, P.B. and W.M.; Funding acquisition, W.M.; Investigation, P.B. and W.M.; Methodology, P.B. and W.M.; Project administration, P.B. and W.M.; Resources, P.B.; Software, P.B.; Supervision, W.M.; Validation, P.B. and W.M.; Visualization, P.B. and W.M.; Writing—original draft, P.B. and W.M.; Writing—review & editing, P.B. and W.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Publicly available pcap file data set of Malware Capture Facility Project was analyzed in this study. It can be found here: https://www.stratosphereips.org/datasets-malware, accessed on 26 March 2021. The pcap file data set provided by CERT Polska cannot be shared due to legal, confidentiality, and privacy issues.

Acknowledgments: The authors would like to thank Paweł Pawliński (CERT Polska/NASK) for his insightful comments and suggestions that allowed for the improvement of this work.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Appendix A. List of Source Code Versions of the Analyzed Fingerprinting Tools

FATT:
https://github.com/0x4D31/fatt/commit/314cd1ff7873b5a145a51ec4e85f6107828a2c79, accessed on 26 March 2021
p0f3plus:

https://github.com/FlUxIuS/p0f3plus/commit/748cc69cc996e830f258f3f3c7b95ca7a4a74a3e, accessed on 26 March 2021
Mercury:
https://github.com/cisco/mercury/commit/500f5b74a710c0f1c423b8cb370c667aae44a7e3, accessed on 26 March 2021

Appendix B. List of Installed Benign Applications Used in Experiments

- Adobe Reader
- Discord
- GIMP
- LibreOffice
- Mozilla Thunderbird
- Notepad++
- Microsoft Office 2019 Home and Business
- Skype
- Slack
- Spotify
- Steam
- VLC media player
- Zoom
- µtorrent

Appendix C. Malware Families of the Final Malware Data Set Sorted by the Number of HTTP Requests

Table A1. Malware families of the final malware data set sorted by the number of HTTP requests in the final data set.

Malware Family Name	Number of Requests	Malware Family Name	Number of Requests
Upatre	62,257	KeyBase	141
Simda	57,730	STOP	139
Locky	44,498	Nessfi	136
Dridex	30,070	Jaff	136
Arkei	22,057	GrayBird	136
DirtJumper	18,486	Cannibal	130
Chthonic	14,410	1ms0rry	129
Vflooder	14,252	IcedID	122
Ursnif	11,756	Wannacry	113
Arid Viper APT	10,063	Adylkuzz	111
Emotet	9662	Amadey	103
Nemucod	8857	ArtraDownloader	99
Houdini	7583	Zeprox	96
Miuref/Boaxxe	7501	PowershellEmpire	88
Pushdo.S	7012	MegalodonHTTP	88
SmokeLoader	6523	BlackshadesRAT	82
Andromeda	5839	Banload	80
Nymaim	5590	GrandSteal	76
Matsnu	5522	Mokes	73
LokiBot	4415	EightRed	73
Kovter.B	4332	ZeroHTTP	70
Tinba	4004	Sakula	67
Formbook	3496	NetSupport	65
AgentTesla	3052	Legion	62
Gaudox	2880	FindPOS	60
BlackNET	2822	DDI.Bot	59
AZORult	2057	Agent.ZJL	57
Mydoom	1833	Adware.Liuliangbao.A	55
Htbot	1730	DCRS	54
Neutrino	1697	Dalexis	52

Table A1. *Cont.*

Malware Family Name	Number of Requests	Malware Family Name	Number of Requests
Kronos	1692	FTCode	50
PUP.Linkury	1481	MSIL.adv	47
Trickbot	1255	Maze	46
Necurs	1158	KPOT	45
Sage	1145	Sality	41
Hancitor	1034	Madness	41
CryptoWall	613	Dimnie	38
Pony	607	Instagram Like Bot	37
Wizzcaster	567	H1N1	36
QuantLoader	538	Panda	35
TVRat	436	Ratankba	34
Kelihos.F	406	Zeroaccess	33
MedusaHTTP	403	DownloadGuide	33
Karmen	397	Betabot	31
GuLoader	383	Alina.POS	31
KINS	351	SocStealer	30
Tofsee.AX	338	Sezin	30
Predator The Thief	286	Scarab	30
InstallCapital	274	Golroted.B	30
Terdot	256	Agima.o	30
TinyNuke	250	CobaltStrike	29
ColorFish	242	Philadelphia	28
HawkEye	234	Dapato	27
Sarwent	229	Mole	26
GandCrab	229	TorrentLocker	24
DustySky	200	FusionCore	23
Phorpiex	190	Qadars	20
DirCrypt	174	KrugBOT	20
Alphacrypt	174	JakyllHyde	20
Donvibs	168	HPDefender.B	20
DiamondFox	153		

References

1. Miller, S.; Smith, P. *Rise of Legitimate Services for Backdoor Command and Control*; Technical Report; Anomali: Redwood City, CA, USA, 2017. Available online: https://www.anomali.com/files/anomali-labs-reports/legit-services.pdf (accessed on 27 July 2020).
2. Broder, A.Z. Some applications of Rabin's fingerprinting method. In *Sequences II*; Capocelli, R., De Santis, A., Vaccaro, U., Eds.; Springer: New York, NY, USA, 1993; pp. 143–152.
3. Li, K.; Chen, R.; Gu, L.; Liu, C.; Yin, J. A Method Based on Statistical Characteristics for Detection Malware Requests in Network Traffic. In Proceedings of the 2018 IEEE Third International Conference on Data Science in Cyberspace (DSC), Guangzhou, China, 18–21 June 2018; pp. 527–532.
4. Perdisci, R.; Lee, W.; Feamster, N. Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces. In Proceedings of the NSDI '10: 7th USENIX Symposium on Networked Systems Design and Implementation, San Jose, CA, USA, 28–30 April 2010; USENIX Association: San Jose, CA, USA, 2010; pp. 391–404.
5. GmbH, A.T. *AV-TEST Security Report 2019/2020*; Technical Report; AV-TEST Institute: Magdeburg, Germany, 2020. Available online: https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2019-2020.pdf (accessed on 15 October 2020).
6. Laperdrix, P.; Bielova, N.; Baudry, B.; Avoine, G. Browser Fingerprinting: A survey. *arXiv* **2019**, arXiv:1905.01051.
7. Bortolameotti, R.; van Ede, T.; Caselli, M.; Everts, M.H.; Hartel, P.; Hofstede, R.; Jonker, W.; Peter, A. DECANTeR: DEteCtion of Anomalous OutbouNd HTTP TRaffic by Passive Application Fingerprinting. In Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC 2017), Orlando, FL, USA, 4–8 December 2017; Association for Computing Machinery: New York, NY, USA, 2017; pp. 373–386. [CrossRef]
8. Bortolameotti, R.; van Ede, T.; Continella, A.; Hupperich, T.; Everts, M.H.; Rafati, R.; Jonker, W.; Hartel, P.; Peter, A. HeadPrint: Detecting Anomalous Communications through Header-Based Application Fingerprinting. In Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC '20), Brno, Czech Republic, 30 March–3 April 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 1696–1705. [CrossRef]
9. Shbair, W.M.; Cholez, T.; Francois, J.; Chrisment, I. A Survey of HTTPS Traffic and Services Identification Approaches. *arXiv* **2020**, arXiv:2008.08339.
10. Stringhini, G.; Egele, M.; Zarras, A.; Holz, T.; Kruegel, C.; Vigna, G. B@bel: Leveraging Email Delivery for Spam Mitigation. In Proceedings of the 21st USENIX Security Symposium (USENIX Security 12), Bellevue, WA, USA, 8–10 August 2012; USENIX Association: San Diego, CA, USA, 2012; pp. 16–32.

11. Bazydło, P.; Lasota, K.; Kozakiewicz, A. Botnet Fingerprinting: Anomaly Detection in SMTP Conversations. *IEEE Secur. Priv.* **2017**, *15*, 25–32. [CrossRef]
12. Fachkha, C.; Bou-Harb, E.; Debbabi, M. Fingerprinting Internet DNS Amplification DDoS Activities. In Proceedings of the 2014 6th International Conference on New Technologies, Mobility and Security (NTMS), Dubai, United Arab Emirates, 30 March–2 April 2014; pp. 1–5. [CrossRef]
13. Kim, T.; Ju, H. Effective DNS server fingerprinting method. In Proceedings of the 2011 13th Asia-Pacific Network Operations and Management Symposium, Taipei, Taiwan, 21–23 September 2011; pp. 1–4. [CrossRef]
14. Blaise, A.; Bouet, M.; Conan, V.; Secci, S. Botnet Fingerprinting: A Frequency Distributions Scheme for Lightweight Bot Detection. *IEEE Trans. Netw. Serv. Manag.* **2020**, *17*, 1701–1714. [CrossRef]
15. Ory, S.; Aharon, F.; Elad, S. *Passive Fingerprinting of HTTP/2 Clients*; Technical Report; Akamai: Cambridge, MA, USA, 2017.
16. Holland, J.; Schmitt, P.; Feamster, N.; Mittal, P. nPrint: Standard Packet-level Network Traffic Analysis. *arXiv* **2020**, arXiv:2008.02695.
17. Białczak, P.; Mazurczyk, W. Characterizing Anomalies in Malware-Generated HTTP Traffic. *Secur. Commun. Netw.* **2020**, *2020*, 8848863. [CrossRef]
18. Sahoo, D.; Liu, C.; Hoi, S.C.H. Malicious URL Detection using Machine Learning: A Survey. *arXiv* **2019**, arXiv:1701.07179.
19. Fowler, G.; Noll, L.C.; Vo, K.P.; Eastlake, D.; Hansen, T. The FNV Non-Cryptographic Hash Algorithm. Available online: https://tools.ietf.org/html/draft-eastlake-fnv (accessed on 27 July 2020).
20. Windows 10 Has an Undocumented Certificate Pinning Feature. Available online: https://web.archive.org/web/20170501183238/http://hexatomium.github.io/2016/09/24/hidden-w10-pins/ (accessed on 27 July 2020).
21. Windows Update—Interception. Available online: https://security.stackexchange.com/questions/31861/windows-update-interception (accessed on 27 July 2020).
22. Arora, J.S. (Ed.) *Introduction to Optimum Design*, 3rd ed.; Academic Press: Boston, MA, USA, 2012.

6 Klasyfikacja rodzin złośliwego oprogramowania przy użyciu protokołu HTTP w scenariuszu otwartozbiorowym

W poniższym rozdziale zaprezentowany zostanie problem klasyfikacji rodzin złośliwego oprogramowania przy użyciu żądań HTTP w scenariuszu otwartozbiorowym. Przedstawione zostaną podstawy teoretyczne problematyki rozpoznania otwartozbiorowego, wykorzystane algorytmy uczenia maszynowego oraz sposób przeprowadzenia badań.

6.1 Wstęp

W celu rozwiązania problemu klasyfikacji otwartozbiorowej złośliwego oprogramowania stworzono kilka podejść bazujących na różnych algorytmach klasyfikacji. Wykorzystano trzy algorytmy otwartozbiorowe: maszyn wektorów nośnych kalibrowanych rozkładem Weibulla (W. J. Scheirer i in. [90]), otwartozbiorowych maszyn wektorów nośnych (P. R. M. Junior i in. [54]) oraz maszyny wektorów ekstremalnych (E. M. Rudd i in. [88]). Zostały one porównane ze standardowymi algorytmami maszyn wektorów nośnych oraz lasów losowych, rozszerzonych o mechanizm odrzucania wyników w sytuacji zbyt niskiego prawdopodobieństwa predykcji. Algorytmy te zostaną zaprezentowane w dalszej części tego rozdziału. Wyniki analiz przedstawionych w poprzednich rozdziałach rozprawy, w tym w szczególności w rozdziale 5., wykorzystano jako źródło definicji cech danych dla badanych algorytmów. W ten sposób dwie reprezentacje żądań HTTP bazujące na narzędziu *Hfinger* zostały porównane z reprezentacją trigramową w celu określenia poziomu wydajności klasyfikacji. Reprezentacja trigramowa bazuje na analizie n-gramowej – klasycznej metodzie przetwarzania języka naturalnego (por. D. Jurafsky i in. [55]). Tworzony jest w niej model częstości występowania w dokumencie lub zbiorze dokumentów wszystkich n-znakowych zbitek – w przypadku trigramu są to zbitki 3 znakowe. W ten sposób tworzony jest wektor liczb, który w łatwy sposób jest przetwarzany przez algorytmy uczenia maszynowego, w przeciwieństwie do oryginalnej tekstowej formy.

Celem tej części rozprawy jest:

- porównanie wydajności klasyfikacji otwartozbiorowej złośliwego oprogramowania dla różnych algorytmów uczenia maszynowego,
- porównanie dwóch rodzajów reprezentacji żądań HTTP narzędzia *Hfinger* pod względem wydajności klasyfikacji otwartozbiorowej złośliwego oprogramowania,

- porównanie wydajności klasyfikacji otwartozbiorowej złośliwego oprogramowania bazującej na reprezentacji żądań HTTP narzędzia *Hfinger* względem reprezentacji trigramowej,
- wybranie najkorzystniejszego algorytmu oraz reprezentacji żądań do celów klasyfikacji pod względem wydajności detekcji, w tym wykorzystania zasobów sprzętowych.

Natomiast główny wkład badawczy tego rozdziału to:

- przystosowanie do jednorodnego interfejsu API popularnych algorytmów otwartozbiorowych oraz ich analiza pod kątem wydajności klasyfikacji i czasu koniecznego do obliczeń na tle tradycyjnych algorytmów uczenia maszynowego,
- przeprowadzenie analizy porównawczej reprezentacji żądań HTTP bazujących na narzędziu *Hfinger* oraz trigramach,
- ocena przydatności badanych algorytmów oraz reprezentacji żądań do klasyfikacji złośliwego oprogramowania w scenariuszu otwartozbiorowym.

6.2 Problematyka rozpoznawania otwartozbiorowego

W poniższym rozdziale zamieszczono opis najważniejszych pojęć stosowanych w rozpoznawaniu otwartozbiorowym (Open Set Recognition), a także jego formalną definicję.

C. Geng i in. wprowadzili podział na cztery kategorie klas w systemach rozpoznawania [42]:

- klasy znane-znane (known known classes), w których dostępne są oznaczone jako pozytywne próbki treningowe z jednoznacznymi etykietami klas, a także informacjami pobocznymi, jak np. informacjami o atrybutach,
- klasy znane-nieznane (known unknown classes), które są oznaczone negatywnie w trakcie treningu, ale nie muszą być znane ich dokładne przyporządkowania do klas lub do grup,
- klasy nieznane-znane (unknown known classes), dla których nie ma próbek w trakcie treningu, ale są dostępne semantyczne informacje poboczne lub informacja o atrybutach,
- klasy nieznane-nieznane (unknown uknown classes), o których nie ma żadnych informacji w trakcie treningu.

W niniejszej rozprawie, ze względu na prostotę, przyjęto nazywać klasy znane-znane klasami znanymi, natomiast klasy nieznane-nieznane klasami nieznanymi. Pozostałe kategorie klas nie są rozpatrywane w niniejszej pracy.

Poniżej znajduje się formalna definicja rozpoznawania otwartozbiorowego bazująca na [17]. Niech f będzie mierzalną funkcją rozpoznania na przestrzeni wejściowej X dla znanej, poprawnej klasy $\hat{V}$. Niech $S_{\hat{V}}$ będzie sumą kul o promieniu r_o, która zawiera wszystkie przykłady treningowe dla wszystkich znanych $x \in \hat{V}$. Niech O będzie przestrzenią otwartą spełniającą zależność $O \subseteq X - S_{\hat{V}}$. Ryzyko otwartoprzestrzenne (open space risk) $R_O(f)$ dla klasy $\hat{V}$ definiowane jest jako:

$$R_O(f) = \frac{\int_O f_{\hat{V}}(x)dx}{\int_{S_{\hat{V}}} f_{\hat{V}}(x)dx} \tag{1}$$

przy czym ryzyko otwartoprzestrzenne jest uznawane za relatywną miarę przestrzeni otwartej (Positively Labeled Open Space - PLOS) w stosunku do całkowitej miary przestrzeni oznaczonej pozytywnie.

Na tej bazie można sformułować następującą definicję:

Definicja 6.1 (Problem rozpoznania otwartozbiorowego)**.** Używając danych treningowych z pozytywnymi próbkami $v_i \in \hat{V}$ oraz innych próbek znanych klas $k_j \in \hat{K}$ oraz miary empirycznego błędu danych/dokładności E, znajdź mierzalną funkcję rozpoznania $f \in H$, gdzie $f(x) > 0$ implikuje pozytywne rozpoznanie dla klasy $\hat{V}$, która minimalizuje błąd otwartozbiorowy:

$$\arg\min_{f \in H} \{R_O(f) + \lambda_r E(f(v_i); f(k_j))\} \tag{2}$$

pod warunkiem, że:

$$m\alpha \leqslant \sum_{i=1}^{m} \phi(f(v_i)), n\beta \geqslant \sum_{j=1}^{n} \phi(f(k_j)) \tag{3}$$

gdzie λ_r określa kompromis regularyzacyjny między ryzykiem otwartoprzestrzennym i ryzykiem empirycznym, gdzie $\alpha \geqslant 0$ i $\beta \geqslant 0$ umożliwiają stworzenie limitu na współczynniki liczby klasyfikacji prawdziwie pozytywnych i/lub fałszywie pozytywnych, $\phi(z)$ jest zdefiniowaną funkcją straty, np. klasyczną zawiasową funkcją straty miękkiego marginesu $\phi(z) = max(0, 1 - z)$ lub kwadratową zawiasową funkcją straty $\phi(z) = max(0, 1 - z)^2$.

Jak twierdzą T.E. Boult i in. [17], minimalnym wymogiem dla algorytmu by był uznany za formalny algorytm rozpoznawania otwartozbiorowego jest spełnienie nierówności $R_O(f) < B$, gdzie B to skończone ograniczenie.

Pewnym rozwiązaniem problemu detekcji otwartozbiorowej może wydawać się rozszerzenie standardowych algorytmów uczenia maszynowego o mechanizm odrzucenia klasyfikacji. W założeniach, wykorzystując estymację prawdopodobieństwa przypisania konkretnej próbki danych do dostępnych klas, można odrzucić wyniki klasyfikacji jeśli maksymalne prawdopodobieństwo jest niższe niż zdefiniowany próg. Prawdopodobieństwo to można interpretować jako pewność klasyfikacji: im jest wyższe, tym wynik jest bardziej pewny. Zatem w sytuacji, gdy klasyfikacja zostaje odrzucona, można uznać, że dana próbka danych należy do klasy nieznanej. Niestety, jak twierdzą T. E. Boult i in. [17], dla większości algorytmów pewność klasyfikacji zwiększa się wraz z oddalaniem od granic decyzyjnych. Dlatego próbka klasy nieznanej mocno oddalona od granic decyzyjnych zostanie oznaczona jako jedna z klas znanych z dużym prawdopodobieństwem. Przez to, że algorytmy te nie mają ograniczonych obszarów decyzyjnych, nie są formalnymi algorytmami rozpoznawania otwartozbiorowego, tj. mają nieograniczoną pozytywnie oznaczoną przestrzeń otwartą, zatem nieskończone ryzyko otwartoprzestrzenne.

6.3 Opis badanych algorytmów uczenia maszynowego

Poniżej zaprezentowano trzy badane algorytmy otwartozbiorowe. Opis zawiera krótkie wyjaśnienie ich działania w tym wpływ wykorzystywanych hiperparametrów. Spośród nich, W-SVM (Weibull calibrated Support Vector Machines) i OSSVM (Open-Set Support Vector Machines) bazują na algorytmie maszyn wektorów nośnych (ang. Support Vector Machines).

6.3.1 Maszyny wektorów nośnych kalibrowane rozkładem Weibulla

Maszyny wektorów nośnych kalibrowane rozkładem Weibulla (Weibull calibrated Support Vector Machines - W-SVM) to algorytm rozpoznawania otwartozbiorowego zaproponowany przez W. J. Scheirera i in. [90]. W artykule autorzy sformułowali model zwartego malejącego prawdopodobieństwa (Compact Abating Probability), w którym to prawdopodobieństwo należenia testowanej próbki do znanej klasy maleje wraz z jej odległością od próbek treningowych. Na bazie tego modelu autorzy zaproponowali algorytm dla dwu etapowej klasyfikacji otwartozbiorowej. W pierwszym etapie wykorzystywany jest model CAP bazujący na jednoklasowej maszynie wektorów nośnych. Jeśli prawdopodobieństwo przynależności testowanej próbki (P_O) będzie mniejsze niż ustalony próg δ_τ, próbka zostanie odrzucona. W drugim etapie wykorzystywany jest klasyfikator binarny, który bazuje na teorii wartości ekstremalnych (Extreme Value Theory). Klasyfikator ten w pierwszym kroku używając rozkładu Weibulla modeluje prawdopo-

dobieństwo przynależności testowanej próbki x do badanej klasy y ($P_\eta(y|x)$). W drugim kroku używając odwrotnego rozkładu Weibulla modeluje prawdopodobieństwo braku przynależności testowanej próbki x do klas negatywnych wobec badanej klasy y ($P_\psi(y|x)$). Wprowadzając zmienną $\iota_y = 1$ jeśli $P_O(y|x) > \delta_\tau$ i $\iota_y = 0$ w przeciwnym wypadku, funkcja rozpoznania wieloklasowego dla klas znanych Y jest zdefiniowana równaniem 4.

$$y^* = \arg\max_{y \in Y} P_\eta(y|x) \times P_\psi(y|x) \times \iota_y \tag{4}$$

pod warunkiem, że

$$P_\eta(y|x) \times P_\psi(y|x) \geqslant \delta_R \tag{5}$$

Progi δ_τ oraz δ_R są hiperparametrami algorytmu W-SVM, ale autorzy zaproponowali, aby przyjąć $\delta_\tau = 0.001$, natomiast δ_R wyznaczać z zależności $\delta_R = 0.5 \times openness$, gdzie $openness$ to otwartość zbioru danych (więcej w 6.4.3).

Kod źródłowy algorytmu W-SVM wykorzystany w badaniach pochodzi z adresu: `https://github.com/ljain2/libsvm-openset`. Kod został dostosowany do API pakietu scikit-learn oraz poprawiono błędy w kodzie uniemożliwiające jego wykonanie, zgodnie z propozycją umieszczoną w oficjalnym repozytorium projektu: `https://github.com/ljain2/libsvm-openset/issues/2`.

6.3.2 Otwartozbiorowe maszyny wektorów nośnych

P. R. M. Junior i in. w [54] zaprezentowali zmodyfikowaną wersję maszyn wektorów nośnych - *Open-Set Support Vector Machines* (OSSVM) - która umożliwia klasyfikację w scenariuszu otwartozbiorowym. Według autorów dla scenariusza klasyfikacji binarnej, w przeciwieństwie do klasycznego algorytmu SVM, OSSVM umożliwia ograniczenie przestrzeni otwartej oznaczonej pozytywnie (Positively Labeled Open Space - PLOS), tzn. przestrzeni otwartej, w której próbki oznaczone są jako pozytywne. Dzięki temu, jeśli OSSVM zostanie użyty w strategii jeden przeciw reszcie (One-vs-Rest) w scenariuszu wieloklasowym, to umożliwi to także ograniczenie przestrzeni otwartej oznaczonej jako znana (Known Labeled Open Space). Jest to przestrzeń niepodparta próbkami treningowymi, a w której próbka testowa byłaby oznaczona jako należąca do jednej ze znanych klas. OSSVM bazuje na optymalizacji punktu obciążenia klasycznych maszyn wektorów nośnych używających gaussowskiej radialnej funkcji bazowej (Radial Basis Function). Jeśli punkt obciążenia jest negatywny, wtedy PLOS jest ograniczony.

Różnica między klasycznym algorytmem maszyn wektorów nośnych a OSSVM widoczna jest przede wszystkim w sformułowaniu problemu pierwotnego separacji hiperpłaszczyzn. Problem pierwotny klasycznych maszyn wektorów nośnych jest często definiowany jak w równaniu 6. Jest to równanie dla miękkiego marginesu określonego parametrem C dla próbek treningowych $\mathbf{x}_i \in \mathbb{R}^d$ i odpowiadającym im etykietom $y_i \in \{-1, 1\}$ dla $i = 1, ..., m$, wektora wag $\mathbf{w}$ oraz zmiennej swobodnej ξ_i.

$$\min_{\mathbf{w},b,\xi} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{m} \xi_i \tag{6}$$

pod warunkiem, że:

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geqslant 1 - \xi_i \; \forall i, \quad \xi_i \geqslant 0 \; \forall i \tag{7}$$

Po odpowiednich przekształceniach[26], otrzymuje się problem optymalizacji jak w równaniu 8.

$$\min_{\alpha} W(\alpha) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{m} \alpha_i \tag{8}$$

pod warunkiem, że:

$$0 \leqslant \alpha_i \leqslant C \; \forall i, \quad \sum_{i=1}^{m} \alpha_i y_i = 0 \tag{9}$$

gdzie α_i to mnożniki Lagrange'a oraz $\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i$.

Natomiast w OSSVM problem pierwotny definiowany jest zgodnie z równaniem 10.

$$\min_{\mathbf{w},b,\xi} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i_1}^{m} \xi_i + \lambda b \tag{10}$$

pod warunkiem, że:

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geqslant 1 - \xi_i \; \forall i, \quad \xi_i \geqslant 0 \; \forall i \tag{11}$$

W rezultacie uzyskany problem optymalizacji podobny jest do algorytmu klasycznego z dokładnością do warunków 9, które przyjmują postać jak w równaniu 12.

$$0 \leqslant \alpha_i \leqslant C \; \forall i, \quad \sum_{i=1}^{m} \alpha_i y_i = \lambda \tag{12}$$

[26]Pełna prezentacja algorytmu maszyn wektorów nośnych wykracza poza ramy niniejszej rozprawy, ale można ją odnaleźć np. w pracy [49].

Według autorów algorytmu OSSVM, parametr λ daje kontrolę nad kompromisem między ryzykiem empirycznym a ryzykiem nieznanego. Jest on uznawany za hiperparametr tego algorytmu.

W badaniach wykorzystano kod źródłowy znajdujący pod adresem: `https://github.com/pedrormjunior/ossvm`. Kod został dostosowany do API pakietu scikit-learn.

6.3.3 Maszyna wartości ekstremalnych

Maszyna wartości ekstremalnych (Extreme Value Machine - EVM) to algorytm rozpoznawania otwartozbiorowego zaproponowany przez z E. M. Rudda i in. w pracy [88]. Autorzy wykorzystują teorię wartości ekstremalnych oraz koncepcję rozkładów marginesów do stworzenia radialnych funkcji włączenia, które modelują prawdopodobieństwa zaliczenia próbki do klas.

Przyjmijmy, że $x \in X$ to próbki treningowe w przestrzeni cech X oraz $y_i \in C \in \mathbb{N}$ to etykieta próbki $x_i \in X$. Rozkład marginesów można zdefiniować jako rozkład wartości odległości od granicy decyzyjnej separującej pary (x_i, x_y), gdzie $\forall j, y_j \neq y_i$ oraz x_j to punkt najbliższy do x_i. Odległość tę można estymować przez $m_{ij} = \frac{\|x_i - x_j\|}{2}$ dla τ najbliższych punktów.

Powyższe wprowadzenie umożliwia prezentację twierdzenia rozkładu marginesów (Margin Distribution Theorem) bazującego na teorii wartości ekstremalnych.

Twierdzenie 1 (Twierdzenie rozkładu marginesów [88])**.** Przyjmijmy, że dana jest pozytywna próbka x_i oraz wystarczająco dużo negatywnych próbek x_j wziętych z dobrze określonych rozkładów klas, dające parami estymaty marginesów m_{ij}. Przyjmijmy, że istnieje ciągły niejednopunktowy rozkład marginesów. Wtedy rozkład wartości minimalnych odległości marginesu dla x_i jest dany rozkładem Weibulla.

Jako że twierdzenie 1 jest prawdziwe dla dowolnego x_i, każdy punkt pozwala estymować rozkład jego odległość do marginesu, tym samym umożliwiając postawienie wniosku 1.

Wniosek 1 (Funkcja gęstości Ψ [88])**.** Przy warunkach zdefiniowanych w twierdzeniu rozkładu marginesów, prawdopodobieństwo, że x' zawiera się w granicy estymowanej przez x_i dane jest wzorem:

$$\Psi(x_i, x'; \kappa_i, \lambda_i) = exp^{-(\frac{\|x_i - x'\|}{\lambda_i})^{\kappa_i}} \tag{13}$$

gdzie $\|x_i - x'\|$ to odległość x' od próbki x_i, a κ_i oraz λ_i to parametry kształtu i skali rozkładu Weibulla otrzymane z dopasowania do najmniejszego m_{ij}.

Funkcja Ψ jest podstawą do określenia czy próbka x' należy do badanej klasy C_l zgodnie z zależnością $\hat{P}(C_l|x') = \arg\max_{i:y_i=C_l} \Psi(x_i, x'; \kappa_i, \lambda_i)$ i jest podstawą dla funkcji decyzyjnej danej równaniem 14.

$$y^* = \begin{cases} \arg\max_{l\in\{1,\ldots,M\}} \hat{P}(C_l|x') & \text{jeśli } \hat{P}(C_l|x') \geqslant \delta, \\ \text{“}nieznana\text{”} & \text{w innych przypadkach.} \end{cases} \tag{14}$$

gdzie δ to próg odrzucenia pomiędzy klasami znanymi i przestrzenią otwartą. W celu lepszej generalizacji wyniku funkcja decyzyjna jest uśredniana po k największych prawdopodobieństwach dla każdej klasy.

Autorzy algorytmu EVM w celu ograniczenia liczby przechowywanych par $\langle x_i, \Psi(x_i, x'; \kappa_i, \lambda_i)\rangle$ zaproponowali, aby wybierać jedynie podzbiór punktów charakteryzujących każdą z klas. W tym celu zdefiniowali strategię minimalizacji, której wynikiem są wektory ekstremalne definiujące ostateczny model. Strategia ta przewiduje wybór tylko tych punktów w klasie, dla których inne punkty tej samej klasy dają prawdopodobieństwo włączenia wyższe niż zadany próg ζ.

Parametry τ, δ, k oraz ζ stanowią hiperparametry algorytmu EVM.

Oryginalny kod źródłowy EVM znajduje się pod adresem: `https://github.com/EMRResearch/ExtremeValueMachine`, natomiast w badaniach wykorzystana została jego zaktualizowana wersja `https://github.com/pedrormjunior/ExtremeValueMachine`, którą dostosowano do API pakietu scikit-learn oraz poprawiono kilka błędów występujących przy przetwarzaniu danych.

6.4 Charakterystyka badań

W poniższym podrozdziale zaprezentowano przeprowadzone badania. Przedstawione zostały użyte zbiory danych oraz protokół badań, tzn. opis schematu ich wykonania. Omówione zostały także wykorzystane miary efektywności klasyfikacji oraz szczegóły wdrożenia użytego protokołu badań.

6.4.1 Dane badawcze

Do przeprowadzenia badań eksperymentalnych w niniejszym rozdziale wykorzystano zbiory danych opisane w rozdziale 5. Zbiory te zostały dodatkowo sprawdzone pod względem poprawności procesu przypisania nazw rodzin. Ostateczny zbiór danych złośliwego oprogramowania zawiera 121 popularnych rodzin złośliwego oprogramowania, na które składa się 401 544 żądania

HTTP[27]. Zbiór danych z ruchem sieciowym popularnych aplikacji niezłośliwych (przeglądarek, aplikacji biurowych itp.) zawiera 248 180 żądań. Oba zbiory danych zostały ze sobą połączone, przy czym podjęto decyzję by żądania aplikacji niezłośliwych były traktowane jako jedna klasa - dodatkowa względem klas złośliwych aplikacji.

W badaniach wykorzystywano dwie główne reprezentacje danych żądań HTTP: reprezentację bazującą na aplikacji *Hfinger* (por. rozdział 5.) oraz reprezentację opartą na trigramach żądań.

Reprezentacja bazująca na aplikacji *Hfinger* została podzielona na dwa podtypy. W pierwszym wykorzystywane są wszystkie informacje, które *Hfinger* analizuje, zaprezentowane w postaci liczby zmiennoprzecinkowej. W drugim podtypie wykorzystywana jest reprezentacja z domyślnego trybu działania aplikacji *Hfinger*. Użycie tych dwóch reprezentacji ma na celu porównanie jak bardzo wybór cech przeprowadzony przy badaniach nad aplikacją *Hfinger* (opisany w rozdziale 5.) może wpływać na zmiany w efektywności klasyfikacji. Porównanie może odpowiedzieć na pytanie, czy do efektywnej klasyfikacji złośliwego oprogramowania wystarczy ograniczona liczba cech oraz ich typ, czy też konieczne jest wykorzystanie ich wszystkich w typie zmiennoprzecinkowym. Podsumowanie wykorzystanych cech znajduje się w tabeli 6.1.

Nazwa cechy	Typ cechy w pełnej reprezentacji *Hfinger*	Typ cechy w domyślnej reprezentacji *Hfinger*
Długość ścieżki URI	Z	C
Liczba poziomów katalogów w URI	C	C
Średnia długość katalogu URI	Z	C
Rozszerzenie żądanego pliku	tekst	tekst
Długość części ze zmiennymi w URI	Z	n/d
Liczba zmiennych w URI	C	n/d
Średnia długość wartości zmiennych URI	Z	Z
Metoda żądania	tekst	tekst
Wersja protokołu	Z	Z
Reprezentacja kolejności nagłówków żądania	tekst	tekst
Reprezentacja wartości popularnych nagłówków	tekst	tekst
Obecność znaków spoza zbioru ASCII w przesyłanych danych	tekst	tekst
Entropia Shannona przesyłanych danych	Z	Z
Rozmiar przesyłanych danych	Z	Z

Tabela 6.1: Cechy żądań HTTP w dwóch reprezentacjach bazujących na aplikacji *Hfinger*. Z - to liczba zmiennoprzecinkowa, C - to liczba całkowita

Nie wszystkie wykorzystywane cechy są liczbami. Cechy tekstowe: rozszerzenie żądanego

[27]Liczba żądań jest mniejsza o 22 żądania względem badań w rozdziale 5. ze względu na odjęcie od rodziny TVRAT niepoprawnie oznaczonych żądań.

pliku, metoda żądania oraz obecność znaków spoza zbioru ASCII w przesyłanych danych zostały zakodowane binarnie przy użyciu funkcji *BinaryEncoder* pakietu Category Encoders[28]. Reprezentacja kolejności nagłówków żądania oraz reprezentacja wartości popularnych nagłówków zostały zakodowane przy użyciu funkcji skrótu (hashing encoding) za pomocą metody *HashingEncoder* pakietu Category Encoders, każda przy użyciu 16 bitów. Ostatecznie po zakodowaniu pełna reprezentacja *Hfinger* posiadała 54 cechy, a domyślna 52 cechy.

Drugi typ reprezentacji żądań HTTP bazuje na analizie trigramowej. W literaturze światowej wielokrotnie potwierdzono, że wykorzystanie n-gramów jako sposobu reprezentacji danych przy detekcji ataków lub złośliwego oprogramowania jest rozwiązaniem skutecznym (por. np. [106], [53]). W badaniach wykorzystywane są trigramy, jako kompromis między wzrostem wymiarowości danych a skutecznością w poprawnej reprezentacji przekształcanych danych. Badania innych autorów (np. A. Oza i in. [79]) wskazują na istotny wzrost efektywności detekcji przy użyciu co najmniej trigramów. Kodowanie pełnych żądań HTTP razem z przenoszonymi danymi zostało wykonane przy użyciu funkcji *HashingVectorizer*[29] biblioteki scikit-learn i domyślnej liczby 2^{20} cech.

Analizowany zbiór danych nie jest zrównoważony. Liczba próbek dla klas waha się od 20 do 248 180 przy średniej arytmetycznej 5325,61 i medianie 160,50. By zmniejszyć wpływ różnicy w liczbie próbek między klasami, zdecydowano się na zrównoważenie liczebności klas poprzez zastosowanie podpróbkowania (undersampling). Ustalono maksymalny limit na 200 próbek, co stanowi 10 krotność liczebności najmniejszej klasy. Przy treningu z każdej klasy przekraczającej limit liczebnościowy, losowane jest 200 próbek, które są ustalane jako reprezentacja danej klasy. Klasy nie są równoważone przy badaniu efektywności klasyfikacji systemu (testowaniu) w celu zapewnienia odzwierciedlenia oryginalnego rozkładu ich liczności. Podpróbkowanie wykonywane jest przy użyciu metody *RandomUnderSampler*[30] pakietu *Imbalanced learn*[31], będącego rozszerzeniem biblioteki scikit-learn.

[28] `https://contrib.scikit-learn.org/category_encoders/`

[29] `https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.HashingVectorizer.html`

[30] `https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.RandomOverSampler.html`

[31] `https://imbalanced-learn.org`

6.4.2 Protokół badań

Celem badań jest porównanie efektywności klasyfikacji różnych algorytmów uczenia maszynowego w scenariuszu otwartozbiorowym, przy jednoczesnym wyborze optymalnych hiperparametrów dla każdego z algorytmów. Standardowe protokoły badań, takie jak podział zbioru danych na część treningową, walidacyjną i testową lub zagnieżdżona walidacja krzyżowa (nested cross validation), zostały opracowane dla problemu zamkniętozbiorowego i nie mogą do końca sprostać problemowi otwartozbiorowemu. Przykładowo, przy wyborze hiperparametrów żaden z tych protokołów nie ma możliwości wzięcia pod uwagę próbek klas nieznanych. By rozwiązać ten problem autorzy C. Geng i in. zaproponowali w [42] by standardowy protokół podziału zbioru danych na część treningową, walidacyjną i testową rozszerzyć o klasy nieznane, które są dodawane do części walidacyjnej oraz testowej zbioru danych. W ten sposób obecność klas nieznanych umożliwia dobranie hiperparametrów oraz ewaluację modelu. Co ważne, "nieznajomość" klas jest symulowana przez losowanie ich odpowiedniego zestawu spośród zestawu klasy znanych dostępnych w zestawie danych.

W podejściu zaproponowanym przez C. Genga i in. estymacja efektywności modelu bazuje na 5 krotnym powtórzeniu analizy dla różnych podziałów zbioru danych. Niemniej w każdym podziale wybór hiperparametrów bazuje na części walidacyjnej zbioru danych, która jest wybierana jednokrotnie razem z częścią treningową i testową. Może to prowadzić do obciążenia estymatora efektywności klasyfikacji, np. przez wybór takiego podziału danych, dla którego hiperparametry nie będą optymalne. Jak omawia S. Raschka w [84] w rozdziale 4.14. by estymacja efektywności klasyfikatora pozostała nieobciążona, konieczne jest odpowiednie zaprojektowanie protokołu badań, np. przez użycie zagnieżdżonej walidacji krzyżowej, dzięki czemu wybór hiperparametrów algorytmu jest odseparowany od analizy efektywności modelu. Mając to na uwadze, w badaniach prezentowanych w niniejszej rozprawie użyto protokołu, który modyfikuje propozycję C. Genga i in. poprzez wprowadzenie do procedury wyboru hiperparametrów zmodyfikowanego sprawdzianu na odłożonych danych (holdout validation). Zamiarem jest zmniejszenie obciążenia generowanego przez tę procedurę.

Protokół badań został zaprezentowany w algorytmie 1. W pętli dokonywany jest podział zbioru danych na część treningową oraz testową. Pętla wykonywana jest 5 razy i za każdym razem losowany jest zestaw klas nieznanych spośród wszystkich badanych klas o wielkości zależnej od otwartości O_P (por. równanie 22). Pozostałe klasy traktowane są jako klasy znane i dzielone na część treningową i testową w stosunku 80:20 przy zachowaniu stosunku liczebnego próbek

w klasach. Część treningowa jest przekazywana do procedury wyboru hiperparametrów algorytmu. Wybrane hiperparametry służą do stworzenia modelu wykorzystującego całość danych treningowych. Do części testowej, składającej się z klas znanych, dołączane są klasy nieznane wybrane na początku pętli i na tak zbudowanym zestawie danych dokonywana jest ewaluacja klasyfikacji. Do tego celu wybrano dwie miary: F1 oraz współczynnik korelacji Matthewsa, o których więcej informacji można znaleźć w podrozdziale 6.4.3.

Algorytm 1 Pseudokod protokołu badań

Dane wejściowe: zbiór danych, otwartość, klasy, miara_klasyfikacji

Dane wyjściowe: efektywność klasyfikacji

```
liczba klas nieznanych ← otwartość × ||klasy||
estymacja klasyfikacji ← 0
for i = 0 to 5 do
    klasy nieznane ← losuj(liczba klas nieznanych)
    klasy znane ← klasy − klasy nieznane
    trening, test ← zbiór danych[klasy znane]
    test ← test + zbiór danych[klasy nieznane]
    najlepsze parametry ← wybór_hiperparametrów(trening, miara_klasyfikacji)
    trenuj_algorytm(trening, najlepsze parametry)
    klasyfikacja ← klasyfikuj(test)
    estymacja klasyfikacji ← estymacja klasyfikacji + miara_klasyfikacji(klasyfikacja)
end for
efektywność klasyfikacji ← estymacja klasyfikacji/5
```

Procedura wyboru najlepszych hiperparametrów jako danych wejściowych używa części treningowej danych z pętli głównej oraz zdefiniowanej miary efektywności klasyfikacji. Wybór hiperparametrów odbywa się przy wykorzystaniu specjalnego przypadku metody przeszukiwania siatki *HalvingGridSearchCV* dostarczonego przez pakiet scikit-learn[32] oraz zmodyfikowanego sprawdzianu na odłożonych danych, służącego jako walidacja krzyżowa dla tej metody.

Metoda *HalvingGridSearchCV* bazuje na iteracyjnym wyborze kombinacji hiperparame-

[32] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.HalvingGridSearchCV.html

trów[33]. W pierwszym kroku dokonywana jest ewaluacja wszystkich kombinacji hiperparametrów na podzbiorze danych treningowych, po czym odrzucana jest jedna trzecia kombinacji z najgorszymi wynikami. W kolejnym kroku iteracji, tak jak w poprzednim, zwiększany jest podzbiór danych i dokonywana jest ewaluacja pozostałych kombinacji hiperparametrów. Natomiast w ostatnim kroku iteracji wykorzystywana jest całość zadanego zbioru danych i wybierany jest najlepszy zestaw hiperparametrów spośród tych, które zostały. Dzięki zastosowaniu metody *HalvingGridSearchCV* możliwe jest przeszukanie całej zadanej przestrzeni hiperparametrów przy mniejszym zużyciu zasobów obliczeniowych niż w przypadku standardowej metody przeszukiwania siatki.

Zmodyfikowany sprawdzian na odłożonych danych bierze pod uwagę występowanie klas nieznanych, co umożliwia trening przy wykorzystaniu danych z tych klas, co jest niedostępne w standardowym podejściu zamkniętozbiorowym. Występowanie klas nieznanych jest symulowane przez podział zestawu klas znanych na pięć równych części. Dla każdego z pięciu podzestawów, klasy, które należą do takiego podzestawu uznawane są nieznane, natomiast pozostała część (dopełnienie do całości zbioru treningowego pętli zewnętrznej) dzielona jest na część treningową i testową w proporcji 80:20, przy zachowaniu proporcji liczbowych próbek w klasach. Do tak powstałej części testowej dołączane są dane klas nieznanych, uzyskane na początku tej pętli przez podział na pięć części oryginalnych danych. W ten sposób dokonywany jest podział danych treningowych, który jest zastosowany jako mechanizm walidacji krzyżowej w metodzie *HalvingGridSearchCV* wyboru najlepszego zestawu hiperparametrów. Wykorzystanie tak zmodyfikowanego sprawdzianu na odłożonych danych ma za zadanie zmniejszyć obciążenie estymacji efektywności klasyfikacji przy wyborze hiperparametrów.

Przedstawiony protokół używany jest do ewaluacji algorytmów dla różnych poziomów otwartości, która określa jak dużo danych o klasach nieznanych wykorzystywanych jest przy testowaniu modeli. Szczególnym przypadkiem protokołu jest sytuacja gdy otwartość jest zerowa, co oznacza, że badany jest klasyczny scenariusz zamkniętozbiorowy. W takiej sytuacji protokół badań zmienia się tylko w procedurze wyboru hiperparametrów. Metoda *HalvingGridSearchCV* jako mechanizmu walidacji krzyżowej używa wtedy standardowej pięciokrotnej warstwowej walidacji krzyżowej (stratified 5-fold cross validation).

[33] `https://scikit-learn.org/stable/modules/grid_search.html#successive-halving-user-guide`

6.4.3 Miary porównawcze

Analiza porównawcza wydajności badanych algorytmów wykorzystuje dwie miary: F1 oraz współczynnik korelacji Matthewsa (Matthews correlation coefficient - MCC). Ze względu na wieloklasowy charakter danych do obydwu miar zastosowano definicje biorące pod uwagę obecność wielu klas. Dodatkowo przyjęto, że próbki nieznanych klas będą grupowane w jedną klasę zbiorczą i traktowane jako dodatkowa klasa w tych miarach.

Miara F1 bada efektywność klasyfikacji i jest średnią harmoniczną precyzji oraz czułości. Ze względu na wieloklasowy charakter danych, w prezentowanej analizie używana jest wersja *makro* miary F1, zdefiniowana w równaniu 15.

$$F1_{makro} = \frac{1}{|K|} \sum_{k \in K} F1_k, \tag{15}$$

gdzie K to zbiór badanych klas, a $F1_k$ to miara F1 dla klasy k.

$F1_k$ to miara F1 dla klasy k obliczana według wzoru 16, w którym precyzja (Precision) P_k jest zdefiniowana wzorem 17, a czułość (Recall) R_k wzorem 18. TP_k to liczba próbek prawdziwie pozytywnych (True Positive) dla klasy k, FP_k to liczba próbek fałszywie pozytywnych (False Positive) dla klasy k, a FN_k to liczba próbek fałszywie negatywnych (False Negative) dla klasy k.

$$F1_k = \frac{2 \times P_k \times R_k}{P_k + R_k}, \tag{16}$$

$$P_k = \frac{TP_k}{TP_k + FP_k}, \tag{17}$$

$$R_k = \frac{TP_k}{TP_k + FN_k}, \tag{18}$$

Wersja *makro* miary F1 jest średnią arytmetyczną miar F1 dla poszczególnych klas, dzięki czemu wynik każdej klasy wpływa w ten sam sposób na wynik ostateczny. Inną wersją miary F1 dla przypadku danych wieloklasowych jest *mikro*, w której na ostateczny wynik duży wpływ ma rozkład liczbowy próbek w poszczególnych klasach.

Wartości miary F1 mieszczą się w przedziale od 0 do 1, gdzie *0* oznacza, że precyzja lub czułość są zerowe, a *1* oznacza perfekcyjne wartości tych miar. W związku z tym, że w badaniach wykorzystywana jest tylko wersja *makro* miary F1, w dalszej części rozdziału, dla wygody i zwięzłości prezentacji, nie będzie wyszczególniane, że to konkretnie ta wersja miary F1 jest

używana. Zatem poniżej, za każdym razem, gdy wspominana jest miara F1, odnosi się to do jej wersji *makro*.

W prezentowanych badaniach do obliczeń miary F1 wykorzystano kod z biblioteki scikit-learn[34].

Współczynnik korelacji Matthewsa mierzy jakość klasyfikacji binarnej biorąc pod uwagę wszystkie cztery elementy macierzy pomyłek, tzn. liczbę elementów prawdziwie pozytywnych, fałszywie pozytywnych, fałszywie negatywnych i prawdziwie negatywnych. Współczynnik można zdefiniować równaniem 19.

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}. \tag{19}$$

gdzie TP to liczba elementów prawdziwie pozytywnych, FP fałszywie pozytywnych, FN fałszywie negatywnych i TN prawdziwie negatywnych. Współczynnik korelacji Matthewsa przyjmuje wartości od -1 do 1, gdzie *1* oznacza perfekcyjną korelację i predykcję, *0* oznacza losową predykcję, a *-1* odwrotną.

Ze względu na wieloklasowy charakter danych, w prezentowanej analizie wykorzystywana jest specjalnie zmodyfikowana do tego celu wersja współczynnika korelacji zdefiniowana w równaniu 20.

$$MCC = \frac{c \times s - \sum_k^K p_k \times t_k}{\sqrt{(s^2 - \sum_k^K p_k^2) \times (s^2 - \sum_k^K t_k^2)}} \tag{20}$$

gdzie dla macierzy błędu C dla klas K:

- $t_k = \sum_i^K C_{ik}$ to liczba razy, gdy klasa k rzeczywiście wystąpiła,
- $p_k = \sum_i^K C_{ki}$ to liczba razy, w których oznaczono klasę k,
- $c = \sum_k^K C_{kk}$ to liczba próbek prawidłowo oznaczonych,
- $s = \sum_i^K \sum_j^K C_{ij}$ to całkowita liczba próbek.

W prezentowanych badaniach do obliczeń tej miary wykorzystano kod z biblioteki scikit-learn[35].

Dodatkowym pojęciem pomocnym w badaniach nad rozpoznawaniem otwartozbiorowym jest *otwartość* określająca stosunek liczby klas znanych podczas fazy treningu algorytmu do

[34] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html

[35] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.matthews_corrcoef.html

liczby klas znanych i nieznanych występujących w fazie testów. Wykorzystywana jest tu definicja podana przez C. Genga i in. [42] używająca nazwy *openness** i określona równaniem 21.

$$O^* = 1 - \sqrt{\frac{2 \times C_T}{C_T + C_E}} \tag{21}$$

gdzie C_T to liczba klas dostępnych w trakcie fazy treningu, a C_E to liczba klas dostępnych w trakcie fazy testowania algorytmu. W przeprowadzonych analizach przyjęto, że $C_E = C_T + U$, gdzie U to liczba klas nieznanych, dostępnych tylko w trakcie testów. Otwartość przyjmuje wartości w zakresie od 0 do 1, gdzie 0 to problem zamkniętozbiorowy.

Ze względu na sposób podziału zbioru danych w przeprowadzonej analizie, używany jest dodatkowy współczynnik otwartości O_P wyrażony w procentach, będący ilorazem liczby klas nieznanych w fazie testowania U i liczby wszystkich klas dostępnych w zbiorze danych K (równanie 22). Współczynnik ten przyjmuje wartości od 0 do 100%, gdzie 0% to problem zamkniętozbiorowy.

$$O_P = \frac{U}{K} \times 100[\%] \tag{22}$$

6.4.4 Opis badań

Celem przeprowadzonych badań eksperymentalnych było porównanie efektywności klasyfikacji wybranych algorytmów uczenia maszynowego w scenariuszu otwartozbiorowym. Do analizy wybrano: maszyny wektorów nośnych kalibrowane rozkładem Weibulla (W-SVM), otwartozbiorowe maszyny wektorów nośnych (OSSVM) oraz maszynę wartości ekstremalnych (EVM). Ich opis zamieszczono w podrozdziale 6.3. Algorytmy W-SVM i EVM zostały wybrane ze względu na ich użycie w projektach dotyczących bezpieczeństwa (W-SVM [22], EVM [50]), liczne odwołania w literaturze oraz dostępność kodu źródłowego, co umożliwiło sprawne ich dostosowanie. Z kolei OSSVM jest algorytmem stosunkowo nowym (publikacja w 2021 r.), ale także bazującym na algorytmie maszyn wektorów nośnych jak W-SVM, dlatego został dołączony do badań w celach porównawczych.

Dodatkowo do badań dołączono dwa standardowe algorytmy jako punkt odniesienia: maszyn wektorów nośnych (SVM) oraz lasów losowych (Random Forest). Algorytmy te są uznawane za rozsądny wybór inżynierski przy klasyfikacji ponad 100 tysięcy próbek danych np. przez poradnik biblioteki scikit-learn[36]. Oba algorytmy używane są w trybie klasyfikacji wielokla-

[36] https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

sowej z predykcją prawdopodobieństwa klasyfikacji. Dzięki temu możliwe jest zastosowanie mechanizmu odrzucenia klasyfikacji. Gdy prawdopodobieństwo klasyfikacji dla danej próbki jest niższe niż ustalony próg odrzucenia, klasyfikacja jest odrzucana i uznaje się, że dana próbka należy do klasy nieznanej. W ten sposób można stworzyć pewien prosty mechanizm, który może pomóc w wykrywaniu klas nieznanych. Niemniej w tej sytuacji żaden z tych dwóch algorytmów nie rozwiązuje problemu otwartozbiorowego ze względu na brak ograniczenia pozytywnie oznaczonej przestrzeni otwartej, a tym samym ryzyko otwartoprzestrzenne pozostaje nieograniczone [17] (por. podrozdział 6.2). Próg odrzucenia δ_R definiowany jest zgodnie z zależnością $\delta_R = 0.5 \times O^*$, gdzie O^* to otwartość zdefiniowana równaniem 21. Taki sposób definicji progu odrzucenia można znaleźć w literaturze ([90]) i służy on jako próg odrzucenia także w algorytmach otwartozbiorowych: jako hiperparametr δ_R w W-SVM i δ w EVM.

Efektywność klasyfikacji algorytmów badana jest dla różnych poziomów liczby klas nieznanych. Jest ona definiowana zgodnie z równaniem 22 i określa ile procent klas nieznanych, w stosunku do wszystkich klas, znajduje się w zbiorze testowym. Przeprowadzone badania zostały wykonane dla pięciu progów wartości od 0% do 75%. Progi te wraz z odpowiadającymi wartościami otwartości O^* znajdują się w tabeli 6.2.

Otwartość O_P [%]	Otwartość O^* [%]
0,00	0,0
10,00	2,64
25,00	7,35
50,00	18,58
75,00	36,96

Tabela 6.2: Poziomy wartości otwartości O_P przyjęte w badaniach i odpowiadające im otwartości O^*.

W trakcie badań dokonano wyboru hiperparametrów dla każdego z algorytmów bazując na metodzie przeszukania siatki dostarczonej przez *HalvingGridSearchCV* biblioteki scikit-learn. Optymalizacja wyników została przeprowadzona osobno dla miary F1 oraz dla miary MCC, ponieważ w przeciwieństwie do klasycznej metody przeszukiwania siatki, w tym przypadku wybór hiperparametrów dokonywany jest jedynie w oparciu na pojedynczej mierze. Przeszukiwane wartości dla poszczególnych algorytmów to:

1. W-SVM
   (a) C: 10^{-3} - 10^{3},

(b) *gamma*: 10^{-3} - 10^{3}

(c) $\delta_\tau = 0.001$ (przyjęto jako stałe zgodnie z zaleceniem autorów algorytmu),

(d) $\delta_R = 0.5 \times O^*$.

2. EVM

(a) τ: 25, 50, 75, 100, 150, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1500, 2000,

(b) δ: $0.5 \times O^*$

(c) k: 3, 4, 5, 6,

(d) ζ: 0,1, 0,2, 0,3, 0,4, 0,5, 0,6, 0,7.

3. OSSVM

(a) $lambda$: 0,0, 0,1, 0,2, ..., 0,9

(b) C: 10^{-3} - 10^{3},

(c) *gamma*: 10^{-3} - 10^{3}

4. wieloklasowy algorytm SVM z progiem odrzucenia

(a) C: 10^{-3} - 10^{3},

(b) *gamma*: 10^{-3} - 10^{3}

5. wieloklasowy algorytm Random Forest z progiem odrzucenia

(a) liczba estymatorów: 100, 200, 300, 500, 750, 1000, 1500, 2000,

(b) kryterium jakości podziału: gini, entropy,

(c) maksymalna liczba cech: auto, None, log2, 0,4, 0,5, 0,6.

Przy wyborze hiperparametrów posłużono się modelem potokowym (pipeline), który w prosty sposób umożliwia zarządzanie składowymi ciągu procedur przetwarzających dane w celu klasyfikacji. Jako potoku użyto funkcji *Pipeline*[37] dostarczanej przez pakiet *Imbalanced learn*. Ze względu na użycie dwóch różnych typów reprezentacji żądań HTTP: przy użyciu aplikacji *Hfinger* oraz jako trigramów, używane są dwa podobne modele przetwarzania danych.

[37] `https://imbalanced-learn.org/stable/references/generated/imblearn.pipeline.Pipeline.html`

W listingu 2. zaprezentowano uproszczony kod źródłowy modelu potokowego dla reprezentacji danych bazującej na aplikacji *Hfinger*. W pierwszym kroku dokonywane jest podróbkowanie (tylko faza treningu), następnie odrzucane są cechy o stałej wartości dla wszystkich próbek (czyli zerowej wariancji), następnie dane są skalowane, by na końcu zostały przekazane do klasyfikatora.

Listing 2: Uproszczony kod źródłowy modelu potokowego dla reprezentacji danych bazującej na aplikacji *Hfinger*. Funkcje pochodzą z biblioteki scikit-learn.

```
pipe = Pipeline(steps=[
        ("rus", RandomUnderSampler()),
        ("feature_filter", VarianceThreshold()),
        ("scaler", MinMaxScaler()),
        ("clf", Classifier())
])
```

W listingu 3. przedstawiono uproszczony kod źródłowy modelu potokowego dla reprezentacji danych bazującej na trigramach. W pierwszym kroku, choć tylko w fazie treningu, dokonywane jest podróbkowanie (undersampling), następnie odrzucane są cechy o stałej wartości dla wszystkich próbek. W kolejnym kroku dokonywane jest ważenie częstością termów z odwrotną częstością w dokumentach. By dodatkowo zmniejszyć wymiarowość danych, w następnym kroku wybierane jest jedynie k najlepszych cech według wartości F analizy wariancji ANOVA. Parametr k jest traktowany tu jako hiperparametr i jego wartość poszukiwana jest w zakresie: 500, 750, 1000, 1500. W ostatnim kroku dane przekazywane są do klasyfikatora.

Listing 3: Uproszczony kod źródłowy modelu potokowego dla reprezentacji danych bazującej na trigramach. Funkcje pochodzą z biblioteki scikit-learn.

```
pipe = Pipeline(steps=[
        ("rus", RandomUnderSampler()),
        ("feature_filter", VarianceThreshold()),
        ("tfidf", TfidfTransformer()),
        ("kbest", SelectKBest()),
        ("clf", Classifier())
])
```

Badania wydajności algorytmów przeprowadzono na dwóch maszynach. Eksperymenty wykorzystujące reprezentacje bazujące na narzędziu *Hfinger* wykonywano na maszynie kontrolo-

wanej przez system operacyjny Kubuntu 18.04.1, wyposażonej w procesor AMD Ryzen 5 1600 oraz 16GB pamięci RAM. Do badań przeznaczone zostało 10 wątków obliczeniowych, które zostały wykorzystane przy wyborze hiperparametrów dla każdego z algorytmów. Niemniej, nie wszystkie algorytmy wspierają obliczenia wielowątkowe, co powoduje wolniejsze wykonanie badań w fazie treningu i ewaluacji tych algorytmów. Tylko dwa algorytmy: EVM oraz Random Forest, domyślnie wspierają obliczenia wielowątkowe. Algorytmy te wykorzystywały 10 dostępnych wątków w trakcie fazy treningu i ewaluacji. Algorytm OSSVM uruchamiany był w strategii jeden przeciw reszcie, bazującej na klasie *OneVsRestClassifier* pakietu scikit-learn[38]. Klasa ta wspiera obliczenia wielowątkowe, ale w praktyce tylko na etapie treningu klasyfikatora. Niemniej udostępnione zostało jej 10 wątków. Algorytmy W-SVM oraz wieloklasowy algorytm SVM z progiem odrzucenia w ogóle nie obsługują obliczeń wielowątkowych.

Z kolei badania wykorzystujące reprezentację trigramową wykonywane były na maszynie kontrolowanej przez system operacyjny Debian 11 i wyposażonej w 20 procesorów wirtualnych taktowanych z częstotliwością 2992MHz oraz 64GB pamięci RAM. Większa moc obliczeniowa i pamięć operacyjna tej maszyny umożliwiła wydajniejsze wykonanie obliczeń niż na maszynie podstawowej, ponieważ reprezentacja trigramowa operuje na większej liczbie cech niż reprezentacje bazujące na narzędziu *Hfinger*. Powoduje to zwiększone zapotrzebowanie na pamięć RAM do przechowywania danych oraz moc obliczeniową, aby przeprowadzać kalkulacje związane z działaniami wybranych algorytmów.

Wersje oprogramowania i najważniejszych pakietów użytych w badaniach to:

- Python 3.8.12
- scikit-learn 1.0
- numpy 1.21.3
- pandas 1.3.4
- imbalanced-learn 0.8.1
- hfinger 0.2.0

[38] `https://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsRestClassifier.html`

6.5 Wyniki badań

Poniżej zaprezentowano wyniki badań wydajności klasyfikacji pięciu analizowanych algorytmów dla dwóch reprezentacji żądań HTTP bazujących na narzędziu *Hfinger* oraz reprezentacji trigramowej. Miarami wydajności są miary: F1 (w wersji makro) oraz MCC, których opis zawarto w podrozdziale 6.4.3. Dodatkowo zmierzono czasy wykonania poszczególnych etapów analizy dla każdego z algorytmów: wyboru hiperparametrów, treningu klasyfikatora oraz ewaluacji klasyfikacji. Zawarte rezultaty, zgodnie z założonym protokołem badań, przedstawiają wartości uśrednione. Czasy te posłużą jako dodatkowa miara przy porównywaniu algorytmów i będą traktowane jako całościowy cykl analizy.

Ponieważ wybór hiperparametrów algorytmów odbywa się poprzez optymalizację pojedynczej miary wydajności, poniżej zaprezentowano wyniki dla obu badanych miar, ale w dwóch osobnych scenariuszach: maksymalizacji miary F1 oraz maksymalizacji miary MCC.

W celu zaoszczędzenia miejsca, w prezentacji posłużono się skróconymi nazwami algorytmów. Podczas gdy skróty *EVM*, *W-SVM* oraz *OSSVM* zostały już wprowadzone wcześniej, to wieloklasowy algorytm SVM z progiem odrzucenia będzie oznaczany jako *MCR-SVM* - *Multi-Class with Rejection SVM*, natomiast wieloklasowy algorytm lasów losowych z progiem odrzucenia krótko jako *RF - Random Forest*.

6.5.1 Pełna reprezentacja narzędzia *Hfinger*

Wyniki analizy efektywności klasyfikacji pięciu badanych algorytmów dla pełnej reprezentacji żądań HTTP tworzonej przez narzędzie *Hfinger* zaprezentowano w tabeli 6.3 (dla wyboru hiperparametrów względem miary F1) oraz w tabeli 6.4 (dla wyboru hiperparametrów względem miary MCC). Każdy z algorytmów badany był dla różnych poziomów otwartości O_P, co przekładało się na różną liczbę klas nieznanych w zbiorze danych ewaluacyjnych.

Efektywność klasyfikacji badanych algorytmów dla wyboru hiperparametrów względem miary F1 zaprezentowano również na rys. 6.1.

Rys. 6.1 wskazuje na spadek efektywności klasyfikacji wraz ze wzrostem otwartości zbioru danych – poziomy miary F1 obniżają się dla wszystkich algorytmów. Choć dla EVM oraz wieloklasowego algorytmu SVM z progiem odrzucenia (MCR-SVM) widoczny jest wzrost dla otwartości na poziomie 10% względem scenariusza zamkniętozbiorowego ($O_P = 0\%$). Podobnie jest z miarą MCC - dla wszystkich algorytmów widoczna jest tendencja spadkowa wraz ze wzrostem otwartości, jednak spadki te są zdecydowanie większe niż dla F1. Dodatkowo tenden-

Algorytm	EVM		W-SVM		OSSVM		MCR-SVM		RF	
Otwartość O_P [%]	F1	MCC	F1	MCC	F1	MCC	F1	MCC	F1	MCC
0	0,73	0,84	0,83	0,66	0,92	0,97	0,83	0,93	0,77	0,94
10	0,79	0,71	0,80	0,49	0,92	0,69	0,93	0,76	0,69	0,73
25	0,76	0,74	0,80	0,45	0,88	0,76	0,90	0,60	0,52	0,52
50	0,71	0,66	0,72	0,36	0,87	0,76	0,82	0,40	0,30	0,27
75	0,66	0,56	0,65	0,25	0,78	0,59	0,62	0,27	0,24	0,20

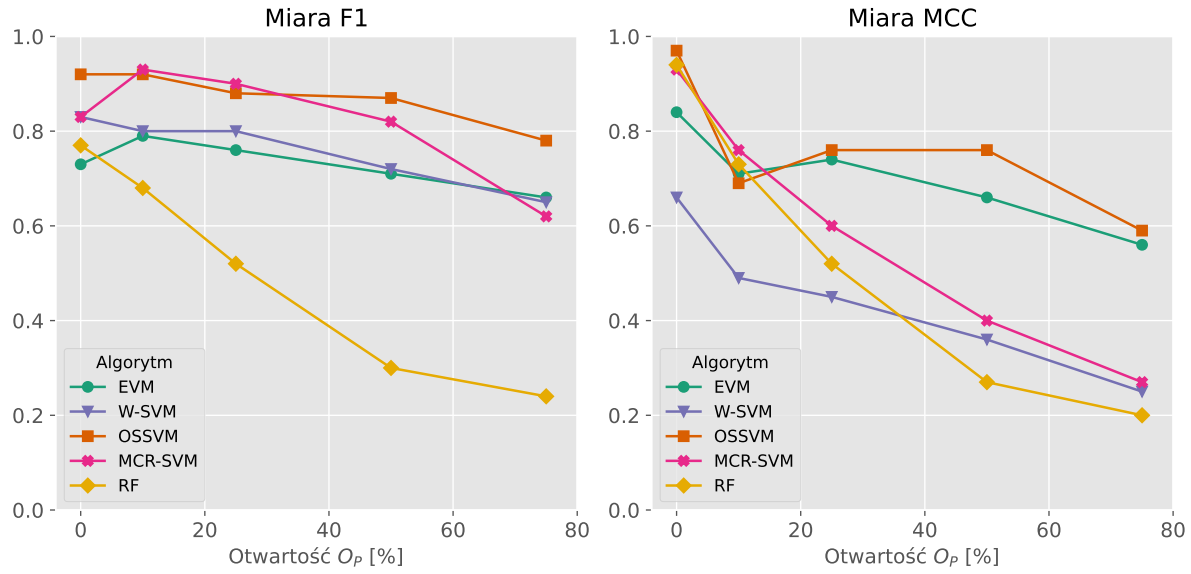
Tabela 6.3: Efektywność klasyfikacji pięciu badanych algorytmów dla pełnej reprezentacji żądań HTTP tworzonych przez narzędzie *Hfinger* dla wyboru hiperparametrów względem miary F1.

Algorytm	EVM		W-SVM		OSSVM		MCR-SVM		RF	
Otwartość O_P [%]	F1	MCC	F1	MCC	F1	MCC	F1	MCC	F1	MCC
0	0,68	0,86	0,62	0,80	0,85	0,96	0,94	0,97	0,77	0,94
10	0,78	0,71	0,62	0,67	0,80	0,73	0,93	0,76	0,67	0,73
25	0,75	0,74	0,58	0,53	0,81	0,82	0,90	0,62	0,52	0,52
50	0,70	0,65	0,41	0,23	0,83	0,78	0,64	0,38	0,30	0,28
75	0,64	0,57	0,54	0,31	0,72	0,66	0,31	0,18	0,25	0,20

Tabela 6.4: Efektywność klasyfikacji pięciu badanych algorytmów dla pełnej reprezentacji żądań HTTP tworzonych przez narzędzie *Hfinger* dla wyboru hiperparametrów względem miary MCC.

cja spadkowa jest zaburzona dla EVM oraz OSSVM. Dla EVM dzieje się tak dla $O_P = 25\%$, a dla OSSVM w przypadku $O_P = 25\%$ i $O_P = 50\%$. Jako że hiperparametry są optymalizowane względem F1, takie przyrosty mogą być wytłumaczone zależnością od liczności rodzin zaliczanych do zestawu danych ewaluacyjnych.

Jak pokazuje rys. 6.1 dla scenariusza zamkniętozbiorowego wszystkie algorytmy osiągają dobre wyniki klasyfikacji, zarówno pod względem miary F1 jak i MCC. Najwyższy poziom F1 - 0,92 - osiągnął OSSVM, zaraz za nim W-SVM i MCR-SVM - z wynikiem 0,83. Algorytmy osiągnęły także dobre poziomy miary MCC - najlepszy był znowu OSSVM (0,97), a zaraz za nim algorytm lasów losowych z progiem odrzucenia (RF) z poziomem 0,94 oraz MCR-SVM z poziomem 0,93. Z kolei algorytm W-SVM uzyskał jedynie 0,66. Wzrost otwartości powoduje spadki efektywności klasyfikacji algorytmów. Spośród nich największe dla algorytmu RF: z poziomu miar F1 i MCC, odpowiednio, 0,77 i 0,94 dla $O_P = 0\%$ do 0,24 i 0,20 dla $O_P = 75\%$, czyli o 0,53 (69%) i 0,74 (79%). Dla innych algorytmów spadek miary F1 od scenariusza $O_P = 0\%$ do $O_P = 75\%$ wahał się od 0,07 (10%) dla EVM, przez 0,14 (15%) dla OSSVM i 0,18

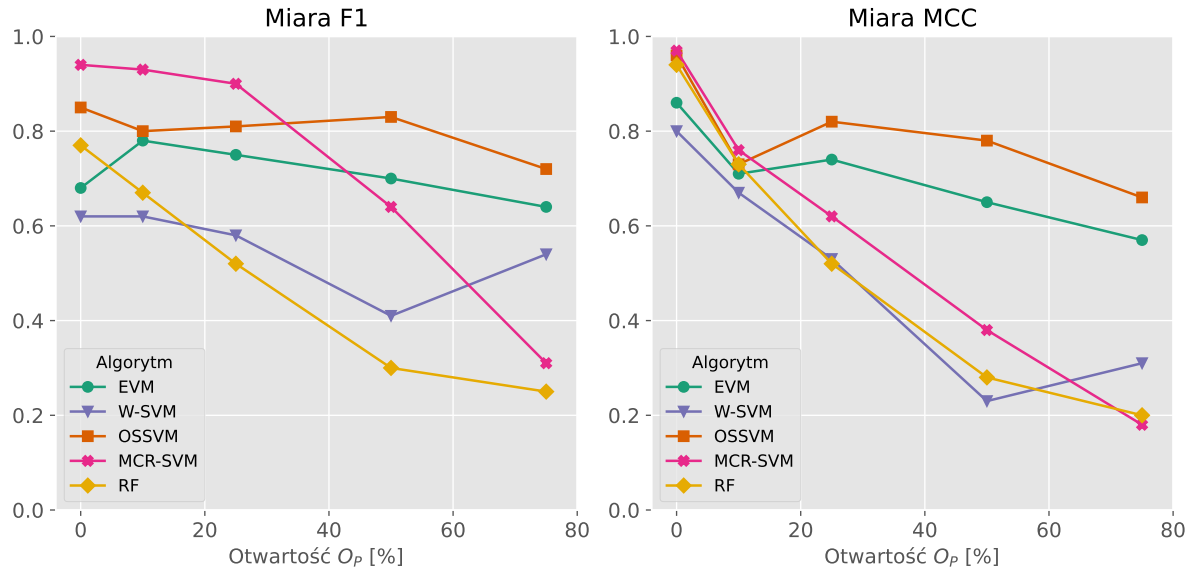


Rysunek 6.1: Efektywność klasyfikacji algorytmów dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger* dla wyboru hiperparametrów względem miary F1.

(22%) dla W-SVM, do 0,21 (25%) dla MCR-SVM. Choć w przypadku algorytmu MCR-SVM maksimum poziomu F1: 0,93 osiągnięte było dla $O_P = 10\%$, co wskazywałoby na jeszcze większą amplitudę F1: 0,31 (33%). Spadki MCC między poziomami $O_P = 0\%$ i $O_P = 75\%$ były większe niż dla F1 dla algorytmów innych niż RF: od 0,28 (33%) dla EVM, przez 0,38 (39%) dla OSSVM i 0,41 (62%) dla W-SVM, do 0,66 (71%) MCR-SVM. Pomimo tych spadków, efektywność klasyfikacji dla poziomów otwartości niższych niż 75% pozostaje akceptowalna dla algorytmów innych niż RF.

Następnie, rys. 6.2 prezentuje efektywność klasyfikacji badanych algorytmów dla wyboru hiperparametrów względem miary MCC.

Rys. 6.2 wskazuje na spadek efektywności klasyfikacji wraz ze wzrostem otwartości zbioru danych, jednak spadek ten jest bardziej zaburzony niż to miało miejsce w scenariuszu wyboru hiperparametrów względem miary F1. W przypadku algorytmów EVM i W-SVM wyższy poziom miary MCC dla scenariusza zamkniętozbiorowego (odpowiednio 0,86 i 0,80) występuje przy zmniejszonych, względem poprzedniego scenariusza wyboru hiperparametrów, poziomach miary F1 - 0,68 i 0,62. Algorytm OSSVM dla $O_P = 0\%$ uzyskał poziom MCC równy 0,96, niższy o 0,01 (1%) niż w poprzednim scenariuszu wyboru hiperparametrów, przy poziomie F1 równym 0,85, niższym o 0,07 (8%). Algorytm MCR-SVM osiągnął lepsze poziomy klasyfikacji niż poprzednio dla $O_P = 0\%$: 0,94 dla F1 i 0,97 dla MCC (wzrost, odpowiednio, o 0,11 (13%)



Rysunek 6.2: Efektywność klasyfikacji dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger* dla wyboru hiperparametrów względem miary MCC.

i 0,04 (4%)). Wynik dla algorytmu RF dla scenariusza zamkniętozbiorowego pozostał na tym samym poziomie: F1 na poziomie 0,77 i MCC na poziomie 0,94. Zatem pod względem F1 i MCC w scenariuszu zamkniętozbiorowym najlepsze okazały się algorytmy MCR-SVM, OSSVM i RF. Najgorszy wynik osiągnął W-SVM - F1 na poziomie 0,62, a MCC na poziomie 0,80.

Podobnie jak to miało miejsce w przypadku wyboru hiperparametrów względem miary F1, algorytmy EVM i OSSVM prezentowały pewne zaburzenie w spadku wydajności klasyfikacji. EVM uzyskał maksimum poziomu F1 (0,78) dla $O_P = 10\%$, po którym nastąpił spadek, co nakłada się na sytuację z poprzedniego scenariusza wyboru hiperparametrów. Odpowiadające wartości F1 i MCC były mniejsze o 0,01 lub równe, choć dla $O_P = 75\%$ miara F1 była mniejsza o 0,02, ale MCC większa o 0,01. W przypadku OSSVM dla $O_P = 25\%$ i $O_P = 50\%$ dochodzi do wzrostu wartości F1 odpowiednio do poziomów 0,81 i 0,83 oraz wzrostu MCC do 0,82 dla $O_P = 25\%$. W przypadku W-SVM widoczny jest spadek wartości F1 i MCC ze wzrostem otwartości, ale dla $O_P = 75\%$ dochodzi do wzrostu ich wartości. Największą różnicę wartości F1 uzyskał algorytm MCR-SVM: 0,63 (67%), za którym plasuje się RF z różnicą 0,52 (68%), EVM z różnicą 0,14 (18%), OSSVM: 0,13 (15%) oraz W-SVM z różnicą 0,21 (34%). Różnice MCC są jeszcze większe: MCR-SVM: 0,79 (81%), RF: 0,74 (79%), W-SVM: 0,49 (61%), OSSVM: 0,30 (31%) i EVM: 0,29 (34%). Najniższy poziom F1: 0,25, uzyskał algorytm RF dla $O_P = 75\%$, najniższy poziom MCC algorytm MCR-SVM: 0,18 dla tego samego poziomu otwartości.

Porównując efektywność klasyfikacji dla scenariuszy wyboru hiperparametrów względem miary F1 i MCC nie sposób uniwersalnie ocenić, który scenariusz jest lepszy. Przykładowo, dla algorytmu MCR-SVM użycie miary MCC powoduje wzrost F1 i MCC dla $O_P = 0\%$, ale pogarsza to znacząco wyniki dla $O_P = 75\%$. W przypadku W-SVM dla poziomów otwartości mniejszych niż 25% wybór hiperparametrów względem MCC poprawia wyniki tej miary kosztem F1, ale powoduje gorsze rezultaty obu miar dla otwartości od 50%. Z tego powodu rozsądne wydaje się wybranie miary, która umożliwia osiągnięcie najlepszych wyników już w konkretnym scenariuszu wdrożenia systemu bazującego na wybranym algorytmie.

Ważnym czynnikiem porównania algorytmów uczenia maszynowego jest czas potrzebny na wykonanie wyboru hiperparametrów, treningu klasyfikatora oraz ewaluacji danych testowych. Systemy, które wymagają dużo czasu na wykonanie tych działań (a tym samym zasobów obliczeniowych), stają się nieefektywne lub nieprzydatne: długi czas oczekiwania na ewaluację zbioru danych może spowodować, że użytkownik zrezygnuje z wykorzystania takiego systemu. Długi czas treningu może natomiast spowodować, że model będzie rzadko aktualizowany, tym samym nie dając odpowiednich predykcji. Wreszcie długi czas wyboru hiperparametrów może powodować, że etap ten nie będzie odpowiednio często wykonywany, tym samym otwierając drogą do używania nieoptymalnych wartości. Będzie to w sposób znaczący wpływać na obniżenie efektywności detekcji. Osobną kwestią, choć niemniej ważną, jest wykorzystanie zasobów sprzętowych: obliczenia, które wymagają więcej czasu na wykonanie, wymagają także więcej energii, tym samym zwiększając koszty operacyjne oraz wywierając negatywny wpływ na środowisko w krajach takich jak Polska, gdzie energia elektryczna produkowana jest głównie ze spalania węgla[39].

Wyniki związane z czasem wykonania wyboru hiperparametrów, treningu klasyfikatora oraz ewaluacji danych testowych dla badanych algorytmów przy wykorzystaniu pełnej reprezentacji żądań HTTP narzędzia *Hfinger* zostały przedstawione w tab. 6.5. Natomiast porównanie graficzne czasu wykonania fazy treningu i ewaluacji dla badanych algorytmów znajduje się na rys. 6.3 przy wyborze hiperparametrów względem miary F1 i na rys. 6.4 dla miary MCC.

[39] https://www.rynekelektryczny.pl/produkcja-energii-elektrycznej-w-polsce/

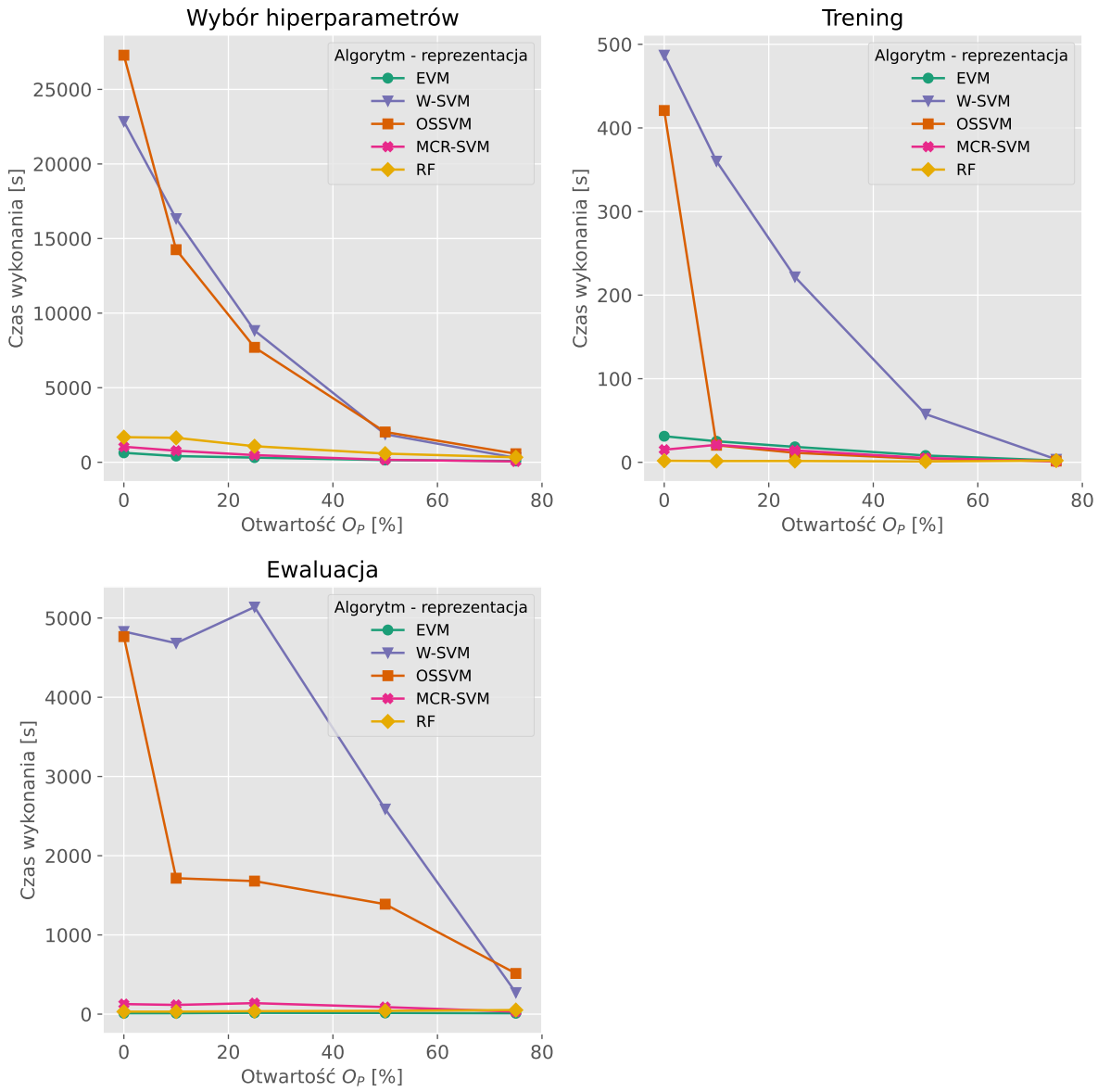
		Optymalizacja F1			Optymalizacja MCC		
Algorytm	O_P [%]	Hiper. [s]	Trening [s]	Ewaluacja [s]	Hiper. [s]	Trening [s]	Ewaluacja [s]
EVM	0	633,71	31,20	11,02	666,91	33,19	9,93
	10	414,66	25,18	11,45	414,56	24,93	11,99
	25	313,59	18,50	16,14	314,97	18,28	15,39
	50	148,28	8,15	14,48	149,40	7,99	17,67
	75	85,86	2,18	10,88	88,86	2,26	9,18
W-SVM	0	22848,78	486,96	4829,60	3202,90	15,56	275,12
	10	16333,31	360,16	4682,41	3749,49	11,85	272,71
	25	8829,98	221,81	5138,29	2076,41	34,13	1376,36
	50	1878,52	57,68	2588,78	524,22	9,35	574,72
	75	297,43	3,58	271,53	143,36	1,88	206,10
OSSVM	0	27291,84	420,82	4765,34	30989,65	422,09	3403,58
	10	14253,77	20,39	1715,70	9340,63	7,56	765,11
	25	7704,17	11,41	1679,43	5883,35	6,87	1374,36
	50	2031,20	3,99	1388,00	1831,03	3,86	1457,24
	75	572,23	1,53	513,65	537,54	1,32	527,08
MCR-SVM	0	1040,92	15,08	126,81	1039,52	25,10	115,56
	10	775,30	20,90	116,57	834,76	20,13	118,30
	25	481,35	14,24	138,72	546,84	18,14	191,08
	50	162,52	5,12	89,17	235,04	14,84	264,19
	75	56,39	1,16	31,43	77,35	4,38	154,00
RF	0	1684,27	1,82	31,73	1732,52	2,89	49,26
	10	1639,77	1,50	31,91	1360,32	1,09	21,30
	25	1076,09	1,55	37,31	955,34	1,11	28,92
	50	581,20	1,06	42,22	513,86	1,77	56,41
	75	334,23	2,05	51,77	251,68	0,73	18,56

Tabela 6.5: Czasy wykonania trzech etapów analizy przez badane algorytmy dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger* (nazwa etapu wyboru hiperparametrów została skrócona do słowa *hiper.* dla zaoszczędzenia miejsca w tabeli).

Przy analizie czasu wyboru hiperparametrów względem miary F1 zaprezentowanych w tab. 6.5 i na rys. 6.3 zauważalny jest jego spadek wraz ze wzrostem poziomu otwartości. Spowodowane jest to zmniejszaniem się liczby klas w danych testowych wraz ze wzrostem otwartości. Najwyższe wartości osiągnęły algorytmy OSSVM: 27291,84s oraz W-SVM: 22848,78s dla $O_P = 0\%$. Najkrótszy czas upłynął dla EVM: 633,71s. MCR-SVM zajęło 1040,92s, a RF:

1684,27s. Zatem czas najwolniejszego algorytmu OSSVM był 43 razy dłuższy niż dla najszybszego EVM. W przypadku $O_P = 75\%$ najdłuższy czas wykonania osiągnął OSSVM: 572,23s i następnie RF: 334,23s. Najszybciej wyboru hiperparametrów dokonał MCR-SVM - w 56,39s. EVM zajęło 85,86s, a W-SVM: 297,43s. W tym przypadku czas najwolniejszego algorytmu OSSVM był 10 razy dłuższy niż dla najszybszego: MCR-SVM.

Wykres czasu fazy treningu na rys. 6.3 wskazuje, że dla $O_P = 0\%$ algorytmy W-SVM i OSSVM osiągały znacząco dłuższe czasy niż trzy pozostałe, np. najwolniejszy W-SVM zajmował 268 razy więcej czasu niż najszybszy: RF. Niemniej jednak dla $O_P = 10\%$ wartość czasu dla OSSVM spadła 21-krotnie. Dla $O_P = 0\%$ jako optymalna wartość hiperparametru λ tego algorytmu wskazana została niezerowa wartość. Wpłynęło to w znaczny sposób także na czasy wyboru hiperparametrów i ewaluacji. Dla pozostałych poziomów otwartości wartość ta była zerowa. Jak to zostało wyjaśnione w sekcji 6.3 parametr λ jest elementem problemu pierwotnego i jego niezerowa wartość w znaczący sposób wpływa na wydłużenie procesu znalezienia rozwiązania problemu optymalizacyjnego SVM. Dla otwartości większych niż 0% W-SVM osiąga najdłuższe czasy spośród badanych algorytmów. Najkrótsze czasy osiąga zaś RF, wynoszą one poniżej 2s - oprócz $O_P = 75\%$, dla którego jest to 2,05s. Analiza czasów ewaluacji ujawnia, że najszybszy pośród wszystkich algorytmów był EVM, osiągający nawet 438 razy mniejszy czas niż najwolniejszy: W-SVM. OSSVM był także stosunkowo wolny - dokonywał ewaluacji w czasie nawet 432 razy dłuższym niż EVM. Algorytm RF był szybszy prawie 4-krotnie niż MCR-SVM dla otwartości innych niż 75% (kiedy to stał się o 65% wolniejszy), niemniej nadal był od 2 do 5 razy wolniejszy niż EVM.

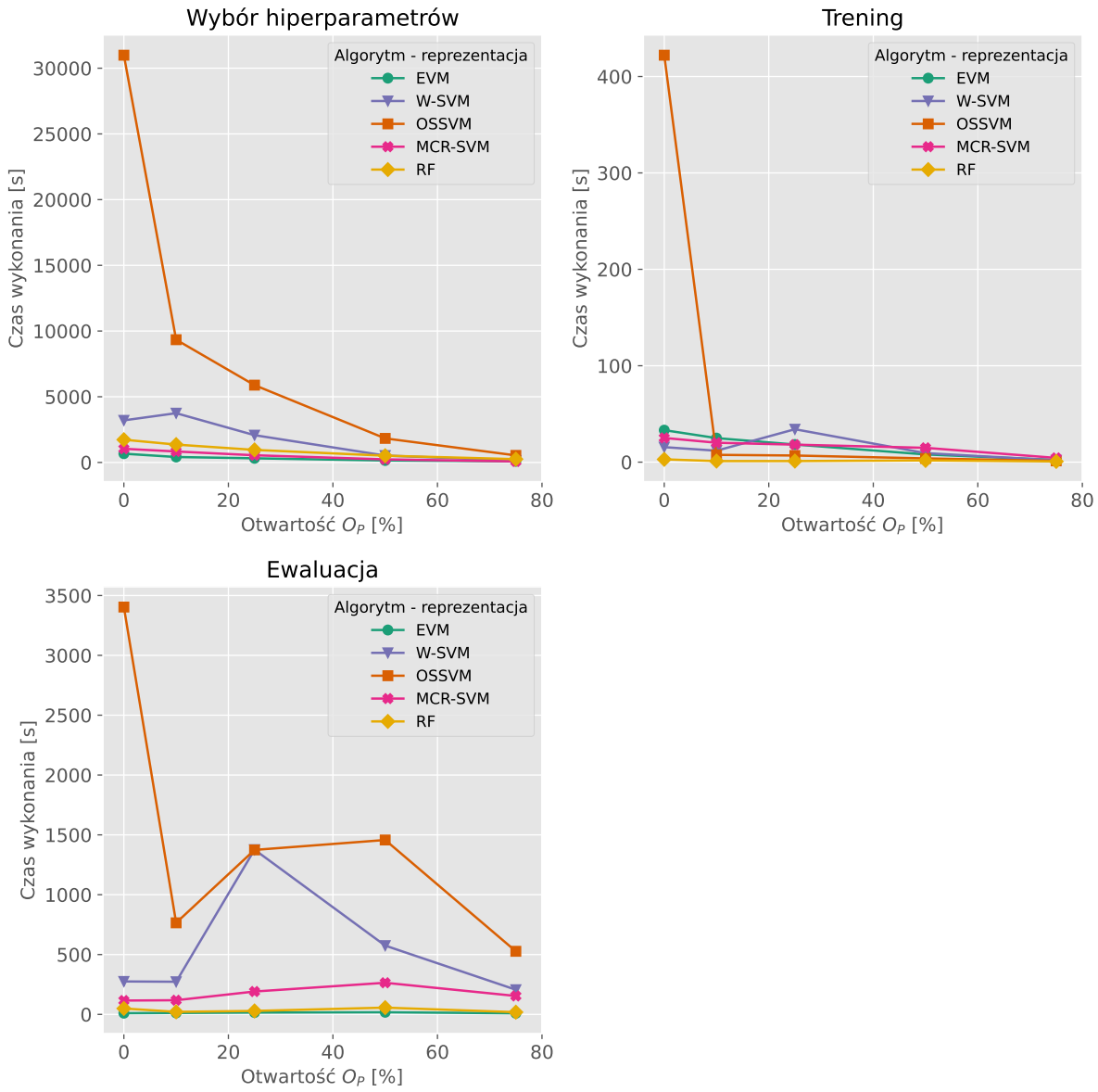


Rysunek 6.3: Czasy wykonania trzech faz analizy przez badane algorytmy dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger*. Wybór hiperparametrów względem miary F1.

Rys. 6.4 prezentuje czasy trzech faz analizy badanych algorytmów przy wyborze hiperparametrów względem miary MCC. Dla wszystkich algorytmów wraz ze wzrostem otwartości malał czas wyboru hiperparametrów, choć dla W-SVM doszło do wzrostu między $O_P = 0\%$ i $O_P = 10\%$ o ok. 547s (17%). Do dość dużego spadku w czasie wyboru hiperparametrów doszło w przypadku OSSVM z 30989,65 s do 9340,63 s (spadek o ok. 70%). Podobnie jak w przypadku wyboru hiperparametrów względem miary F1, było to związane z niezerową wartością hiperparametru λ dla $O_P = 0\%$. Spośród czasów wyboru hiperparametrów względem miary MCC najdłuższy czas dla $O_P = 0\%$ osiągnął algorytm OSSVM: 30989,65s oraz W-SVM:

3202,90s. Najkrótszy czas osiągnął ponownie EVM: 666,91s. MCR-SVM osiągnął 1039,52s, a RF: 1732,52s. Zatem najszybszy algorytm EVM zajmował 46 razy mniej czasu niż najwolniejszy: OSSVM. Dla poziomu otwartości równego 75% najdłuższy czas 537,54s osiągnął OSSVM oraz RF: 251,68s. Najkrótszy czas upłynął dla MCR-SVM: 77,35s. EVM osiągnął 88,86s, a W-SVM: 143,36s. W tym przypadku algorytm OSSVM, jako najwolniejszy, zajął 7 razy więcej czasu niż najszybszy tj. MCR-SVM.

Porównując czasy treningu, algorytm RF znowu osiąga najniższe wyniki spośród wszystkich w zakresie od 0,73s do 2,89s. Z kolei algorytm OSSVM dla $O_P = 0\%$ uzyskuje najwyższą wartość spośród wszystkich algorytmów: 422,09s, niemniej dla kolejnego poziomu otwartości jest to już 7,56s, czyli 56 krotny spadek. Dla otwartości innych niż 0% OSSVM charakteryzują czasy krótsze niż dla innych algorytmów (oprócz najszybszego RF), w tym W-SVM. Ten ostatni dla $O_P = 0\%$ i $O_P = 10\%$ osiąga czasy ok. 2-krotnie niższe niż algorytm EVM i MCR-SVM, jednak dla $O_P = 25\%$ dochodzi do 3-krotnego wzrostu czasu. Dla otwartości aż do 25% włącznie MCR-SVM jest szybszy o co najwyżej 24% niż EVM, ale dla wyższych wartości otwartości sytuacja się odwraca. Analiza czasu ewaluacji wskazała, że algorytmy W-SVM oraz OSSVM nadal są wolniejsze niż pozostałe trzy algorytmy. Dla $O_P = 0\%$ najwolniejszy OSSVM dokonuje ewaluacji w czasie 343 razy dłuższym niż najszybszy EVM. W-SVM tylko dla $O_P = 25\%$ uzyskuje czas powyżej 1000s, niemniej czasy te są nadal dłuższe niż dla MCR-SVM czy RF. EVM jest szybszy od MCR-SVM w zakresie od 10 do 17 razy, natomiast od RF w zakresie od 2 do 5 razy. RF jest także szybszy od MCR-SVM w zakresie od 2 do 8 razy.



Rysunek 6.4: Czasy wykonania trzech faz analizy przez badane algorytmy dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger*. Wybór hiperparametrów względem miary MCC.

Porównując czasy wykonania analizy pod względem miary wykorzystanej do wyboru hiperparametrów, można wyciągnąć wnioski zależne od algorytmu. Dla algorytmu EVM nie widać znaczących zmian w czasach wykonania między tymi scenariuszami. Są one jednak znaczące dla algorytmu W-SVM – przy wyborze miary MCC do wyboru hiperparametrów znacząco zmniejszyły się czasy wykonania poszczególnych etapów analizy: dla $O_P = 0\%$ czas wyboru hiperparametrów zmniejszył się 7-krotnie, czas treningu 31-krotnie, a ewaluacji 17-krotnie. Powodem tej zmiany jest użycie innych wartości hiperparametru γ: przy optymalizacji względem miary F1 parametr ten przyjmował najczęściej wartość 10^3, natomiast przy optymalizacji względ-

dem miary MCC przyjmował wartości mniejsze od zera. Widoczne jest to także w zwiększeniu czasu treningu i ewaluacji dla $O_P = 25\%$ przy wyborze hiperparametrów względem miary MCC: wzrost względem niższego poziomu otwartości spowodowany jest wyborem wartości 10^3 jako parametru γ dla jednej z wewnętrznych iteracji protokołu analizy. Doszło w niej do znaczącego wzrostu czasu wykonania w stosunku do innych iteracji dla tego poziomu otwartości, co wpłynęło na zwiększenie zaraportowanego, uśrednionego czasu wykonania. Zależność czasu wykonania analizy od wartości hiperparametrów widoczna jest również w przypadku algorytmu OSSVM. Tak jak algorytm W-SVM, bazuje on na algorytmie maszyn wektorów nośnych z radialną funkcją bazową, zatem także wykorzystuje hiperparametry C oraz γ. Porównując czasy wykonania, największe różnice widać dla otwartości w zakresie od 0% do 25%, co spowodowane jest wyborem wartości większych niż 100 zarówno dla hiperparametru C, jak i γ dla optymalizacji względem F1, podczas gdy dla optymalizacji względem MCC wartości te były niższe niż 10. Do podobnych obserwacji można dojść dla algorytmu MCR-SVM także bazującego na algorytmie maszyn wektorów nośnych z radialną funkcją bazową - różnice w czasie wykonania przy optymalizacji miarami F1 i MCC podyktowane są różnymi wartościami hiperparametrów. W związku z tym, że czasy wykonania są zdecydowanie krótsze niż np. dla OSSVM, różnice te nie są aż tak widoczne w wartościach absolutnych. W przypadku algorytmu RF różnice między czasami wykonania analizy są zależne od hiperparametru liczby estymatorów: w sytuacji gdy liczba ta się zwiększa, rośnie także czas wykonania analizy - jak ma to miejsce np. dla czasu estymacji dla $O_P = 0\%$. Niemniej różnice w wartościach absolutnych nie są aż tak duże, jak w przypadku algorytmu OSSVM.

Zaprezentowane powyżej wyniki można wykorzystać do oceny analizowanych algorytmów pod względem przydatności w klasyfikacji złośliwego oprogramowania w scenariuszu otwartozbiorowym. Pod względem efektywności klasyfikacji wyróżniają się dwa algorytmy: OSSVM oraz MCR-SVM. Oba osiągają dobre wyniki w scenariuszu zamkniętozbiorowym, lepsze niż pozostałe algorytmy, zarówno jeśli chodzi o miarę F1, jak i MCC. Przy wyborze hiperparametrów względem miary F1, algorytm OSSVM osiąga lepszy niż algorytm MCR-SVM poziom miar F1 i MCC dla $O_P = 0\%$ (odpowiednio o 0,09 [11%] i 0,04 [4%]) oraz gorsze dla $O_P = 10\%$ (odpowiednio 0,01 [1%] i 0,07 [9%]). Niestety wraz ze wzrostem otwartości już od 25% w algorytmie MCR-SVM dochodzi do znaczących spadków miary MCC, większych niż w OSSVM. W przypadku wyboru hiperparametrów względem miary MCC, algorytm OSSVM dla otwartości $O_P = 0\%$ i $O_P = 10\%$ ma gorsze poziomy miary F1 o, odpowiednio, 11% i 16% niż algorytm

MCR-SVM oraz niższe poziomy MCC o 1% i 4%. Jednak dla otwartości na poziomie 25%, mimo mniejszej o 10% wartości F1, OSSVM osiąga o 32% wyższą wartość MCC. Dla otwartości równej co najmniej 50% miary F1 są większe dla OSSVM o 30% i 132% odpowiednio dla $O_P = 50\%$ i $O_P = 75\%$, oraz w tej samej kolejności 105% i 267% dla miary MCC.

Algorytm OSSVM osiąga najlepsze poziomy klasyfikacji spośród algorytmów otwartozbiorowych dla wszystkich poziomów otwartości, ale obliczenia, których wymaga, zajmują sumarycznie nawet 49 razy więcej czasu niż dla algorytmu EVM, a w przypadku wyboru hiperparametrów względem miary MCC, nawet 10 razy dłużej niż dla algorytmu W-SVM. Algorytm OSSVM bazuje na strategii *jeden przeciw reszcie* (One versus Rest), co wymusza tworzenie osobnego podklasyfikatora dla każdej klasy znanej i przetwarzanie danych przy użyciu każdego z takich podklasyfikatorów. Taka architektura zwiększa złożoność głównego klasyfikatora, a także wydłuża czas obliczeń. By skrócić ten czas, można wykorzystać mechanizm zrównoleglenia obliczeń, co wykorzystano przy analizie tego algorytmu w fazie wyboru hiperparametrów oraz treningu. Stworzenie metaklasyfikatora obsługującego zrównoleglenie obliczeń na etapie ewaluacji przy wykorzystaniu klasy *OneVsRestClassifier* pakietu scikit-learn mogłoby skrócić ten etap. Jednak przy prostym założeniu 10-krotnego skrócenia czasu ewaluacji, OSSVM byłby nadal w niej wolniejszy od EVM od ok. 5 do ponad 400 razy.

Porównanie efektywność klasyfikacji algorytmów EVM i W-SVM pokazuje, że dla wyboru hiperparametrów ze względu na miarę F1, W-SVM osiąga wyższe poziomy miary F1 o od 1% ($O_P = 10\%$) do 14% ($O_P = 0\%$). Jednak dzieje się to przy znacząco niższych poziomach MCC o od 21% dla $O_P = 0\%$ do 55% dla $O_P = 75\%$. W przypadku wyboru hiperparametrów względem miary MCC algorytm EVM osiąga wyższe poziomy klasyfikacji od W-SVM: dla F1 o od 10% do 71%, a dla MCC o od 6% do 183%. Przy porównaniu czasów wykonania analizy faworytem jest EVM. Przy wyborze hiperparametrów względem miary F1, EVM jest szybszy w każdym z trzech etapów analizy: od 3 do 39 razy przy wyborze hiperparametrów, od ok. 2 do 16 razy przy treningu i od 25 do 438 razy przy ewaluacji. Przy wyborze hiperparametrów względem miary MCC, EVM jest nadal szybszy przy wyborze hiperparametrów (od 2 do 9 razy) oraz przy ewaluacji (od 22 do 89 razy), choć dla niektórych poziomów otwartości ($O_P = 0\%$ i $O_P = 10\%$) jest dwukrotnie wolniejszy przy treningu.

W porównaniu do innych algorytmów, algorytm RF ma zadowalające poziomy klasyfikacji tylko dla scenariusza zamkniętozbiorowego, choć wtedy algorytmy OSSVM lub MCR-SVM mają lepszą efektywność klasyfikacji nawet o, odpowiednio, 19% i 22% dla F1 oraz 3% dla

MCC w przypadku OSSVM. Choć dla wyboru hiperparametrów względem F1, RF osiąga wartość MCC o 1% wyższą niż MCR-SVM. Dla pozostałych poziomów otwartości algorytm ten ma niższe poziomy miar F1 lub MCC niż pozostałe algorytmy. Algorytm RF ma jednak krótsze czasy analizy danych niż algorytmy OSSVM i W-SVM, a dla fazy treningu znacząco krótsze niż pozostałe algorytmy z dokładnością do $O_P = 75\%$, kiedy to czasy dla MCR-SVM i OSSVM mogą być krótsze.

Biorąc pod uwagę przedstawione powyżej cechy badanych trzech algorytmów otwartozbiorowych, rozsądnym wyborem przy klasyfikacji złośliwego oprogramowania w scenariuszu otwartozbiorowym jest algorytm EVM, posiadający zadowalające poziomy efektywności klasyfikacji oraz krótkie czasy wykonania analizy. Spośród wszystkich badanych algorytmów dobrym wyborem jest MCR-SVM zapewniający dobre poziomy klasyfikacji, a pod względem miary F1 lepsze niż EVM dla poziomu otwartości do 25% włącznie, choć gorsze poziomy MCC dla poziomów otwartości także od 25%. Jeżeli stałoby się możliwe znaczące skrócenie czasu wykonania analiz algorytmu OSSVM, to byłby on prawdopodobnie najlepszym wyborem spośród wszystkich badanych.

6.5.2 Domyślna reprezentacja narzędzia *Hfinger*

Do analizy wydajności klasyfikacji pięciu wybranych algorytmów uczenia maszynowego wykorzystano również drugi typ reprezentacji żądań HTTP bazujący na narzędziu *Hfinger*. Jest to tryb domyślny działania tego narzędzia, który był wynikiem wyboru cech przedstawionego w rozdziale 5. niniejszej rozprawy. Tryb domyślny wykorzystuje podzbiór zestawu cech zawartych w pełnej reprezentacji opisanej w poprzednim rozdziale. Porównanie wyników klasyfikacji tych dwóch reprezentacji ma odpowiedzieć na pytanie, czy do efektywnej klasyfikacji złośliwego oprogramowania wystarczy ograniczona liczba cech oraz ich typ, czy raczej konieczne jest wykorzystanie ich wszystkich. Pośrednio umożliwi to także ocenę wyboru cech dokonanego przy tworzeniu narzędzia *Hfinger*.

Wyniki analizy wydajności klasyfikacji badanych algorytmów przy wykorzystaniu domyślnej reprezentacji żądań HTTP narzędzia *Hfinger* przedstawiono w tab. 6.6 - dla wyboru hiperparametrów względem miary F1, oraz w tab. 6.7 - dla wyboru hiperparametrów względem miary MCC.

Natomiast tab. 6.8 zawiera czasy wykonania trzech etapów analizy badanych algorytmów.

Wyniki klasyfikacji domyślnej reprezentacji żądań HTTP narzędzia *Hfinger* są w dużym

Algorytm	EVM		W-SVM		OSSVM		MCR-SVM		Random Forest	
Otwartość O_P [%]	F1	MCC	F1	MCC	F1	MCC	F1	MCC	F1	MCC
0	0,72	0,85	0,66	0,65	0,92	0,97	0,86	0,95	0,75	0,94
10	0,78	0,71	0,63	0,39	0,91	0,68	0,90	0,76	0,66	0,72
25	0,74	0,72	0,61	0,38	0,87	0,79	0,88	0,62	0,51	0,51
50	0,69	0,65	0,57	0,29	0,84	0,77	0,83	0,43	0,29	0,28
75	0,64	0,55	0,54	0,18	0,73	0,54	0,62	0,28	0,24	0,20

Tabela 6.6: Efektywność klasyfikacji pięciu badanych algorytmów dla domyślnej reprezentacji żądań HTTP tworzonych przez narzędzie *Hfinger* dla wyboru hiperparametrów względem miary F1.

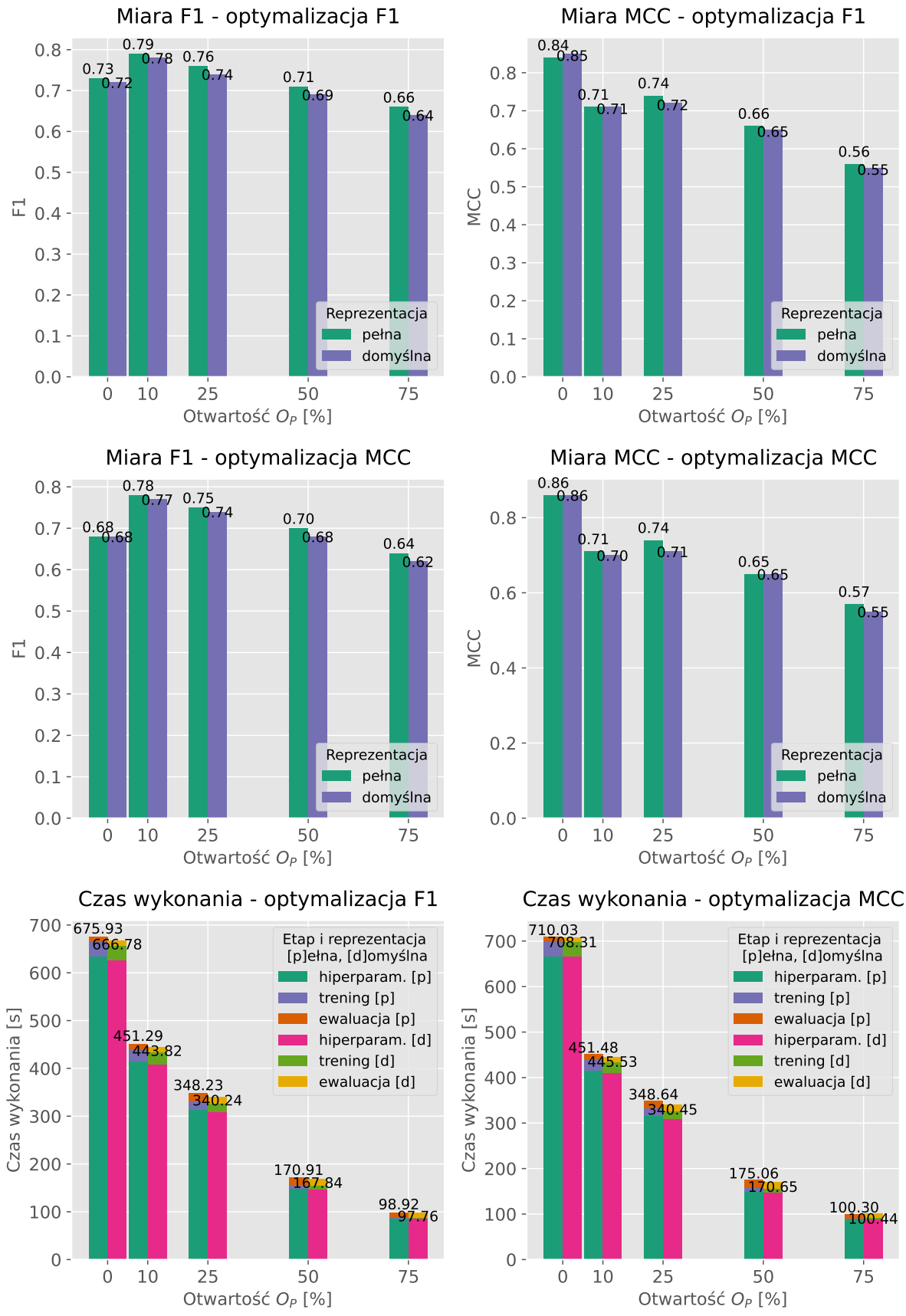
Algorytm	EVM		W-SVM		OSSVM		MCR-SVM		Random Forest	
Otwartość O_P [%]	F1	MCC	F1	MCC	F1	MCC	F1	MCC	F1	MCC
0	0,68	0,86	0,56	0,79	0,90	0,97	0,92	0,97	0,74	0,93
10	0,77	0,70	0,55	0,63	0,79	0,73	0,91	0,76	0,67	0,72
25	0,74	0,71	0,55	0,47	0,78	0,81	0,89	0,62	0,52	0,51
50	0,68	0,65	0,43	0,29	0,82	0,79	0,61	0,36	0,29	0,27
75	0,62	0,55	0,52	0,28	0,66	0,55	0,43	0,33	0,23	0,20

Tabela 6.7: Efektywność klasyfikacji pięciu badanych algorytmów dla domyślnej reprezentacji żądań HTTP tworzonych przez narzędzie *Hfinger* dla wyboru hiperparametrów względem miary MCC.

stopniu podobne do wyników pełnej reprezentacji. W związku z tym, w celu ograniczenia rozmiarów rozprawy oraz jasności wywodu, omówienie wyników domyślnej reprezentacji zostanie pominięte na rzecz poszerzonego porównania wyników obu reprezentacji. W tym celu stworzono rys. 6.5-6.9, na których znajduje się zbiorcze zestawienie wyników klasyfikacji oraz czasów wykonania analizy dla obu reprezentacji żądań narzędzia *Hfinger*. Dla każdego z algorytmów wyniki umieszczono na osobnym rysunku dla ułatwienia porównania ich wartości.

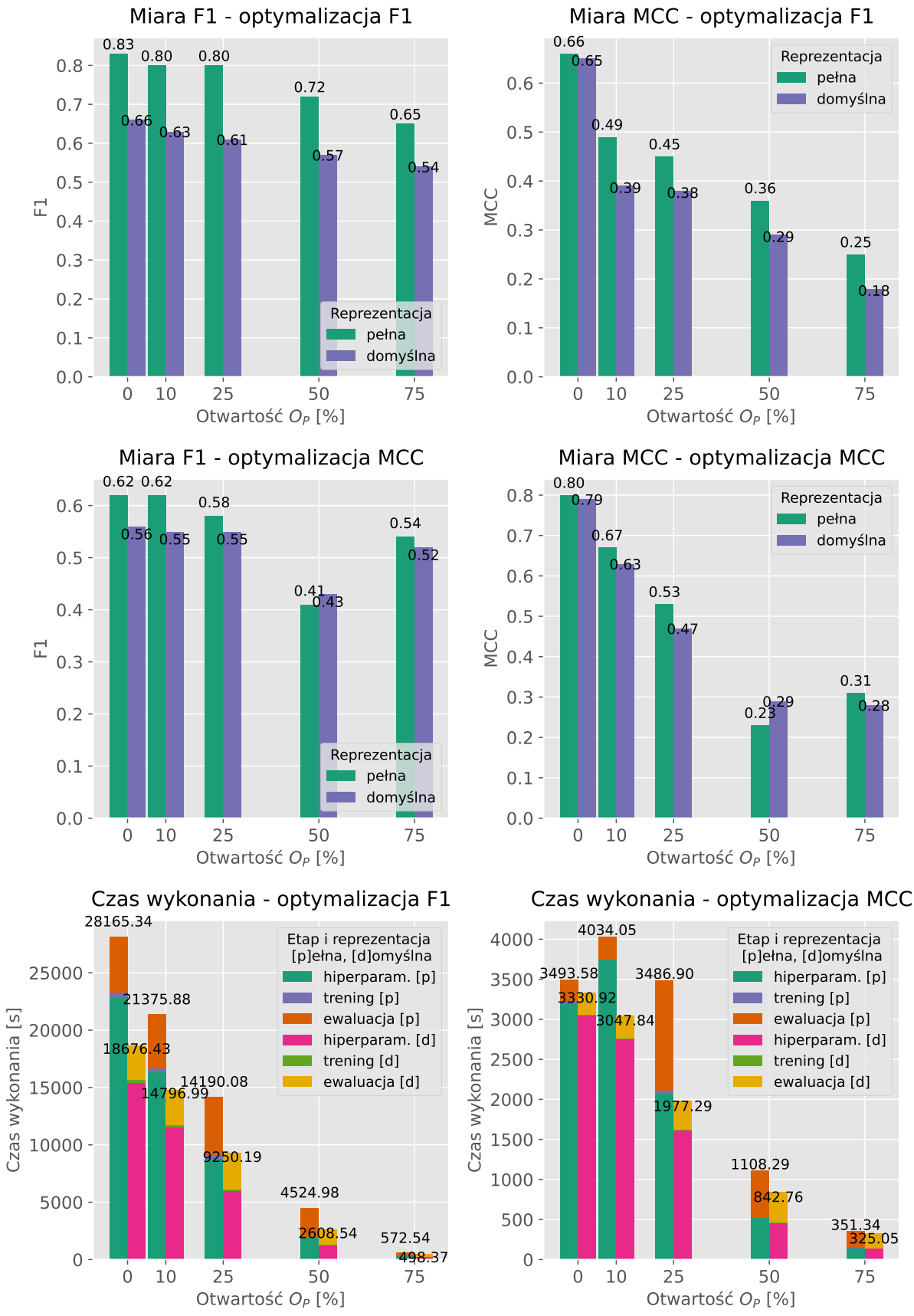
		Optymalizacja F1			Optymalizacja MCC		
Algorytm	O_P [%]	Hiper. [s]	Trening [s]	Ewaluacja [s]	Hiper. [s]	Trening [s]	Ewaluacja [s]
EVM	0	626,40	30,69	9,69	666,22	32,62	9,47
	10	408,40	24,59	10,83	409,74	24,88	10,91
	25	308,52	17,92	13,80	308,67	18,00	13,78
	50	146,41	7,84	13,59	148,01	7,87	14,77
	75	85,56	2,03	10,17	89,17	2,26	9,01
W-SVM	0	15427,78	270,79	2977,86	3045,22	15,06	270,64
	10	11524,26	207,94	3064,79	2753,25	11,69	282,90
	25	5973,48	128,11	3148,60	1611,91	8,50	356,88
	50	1249,34	27,79	1331,41	458,16	5,66	378,94
	75	244,32	3,18	250,87	140,92	1,75	182,38
OSSVM	0	28462,53	297,38	5990,70	30447,79	340,98	5304,18
	10	12173,19	18,48	1779,14	8823,57	7,43	764,01
	25	6730,86	9,98	1695,32	5532,35	6,10	1253,61
	50	1915,22	3,58	1265,17	1833,79	3,36	1245,20
	75	550,80	1,46	495,46	522,16	1,86	581,36
MCR-SVM	0	958,09	15,99	110,96	964,43	20,90	115,91
	10	719,75	17,14	111,60	763,13	20,58	124,24
	25	445,69	12,26	129,99	477,40	13,54	135,04
	50	150,34	5,12	96,72	219,03	18,06	295,47
	75	54,10	1,11	30,41	73,31	4,13	134,59
RF	0	1569,65	0,93	16,53	1611,86	5,21	73,62
	10	1594,03	2,57	52,90	1505,34	2,79	58,44
	25	988,86	2,48	65,19	932,84	1,55	46,65
	50	515,68	1,45	50,33	459,28	0,81	29,14
	75	298,61	0,60	14,73	245,86	1,17	29,62

Tabela 6.8: Czasy wykonania trzech etapów analizy przez badane algorytmy dla domyślnej reprezentacji żądań HTTP narzędzia *Hfinger* (nazwa etapu wyboru hiperparametrów została skrócona do słowa *hiper.* dla zaoszczędzenia miejsca w tabeli).



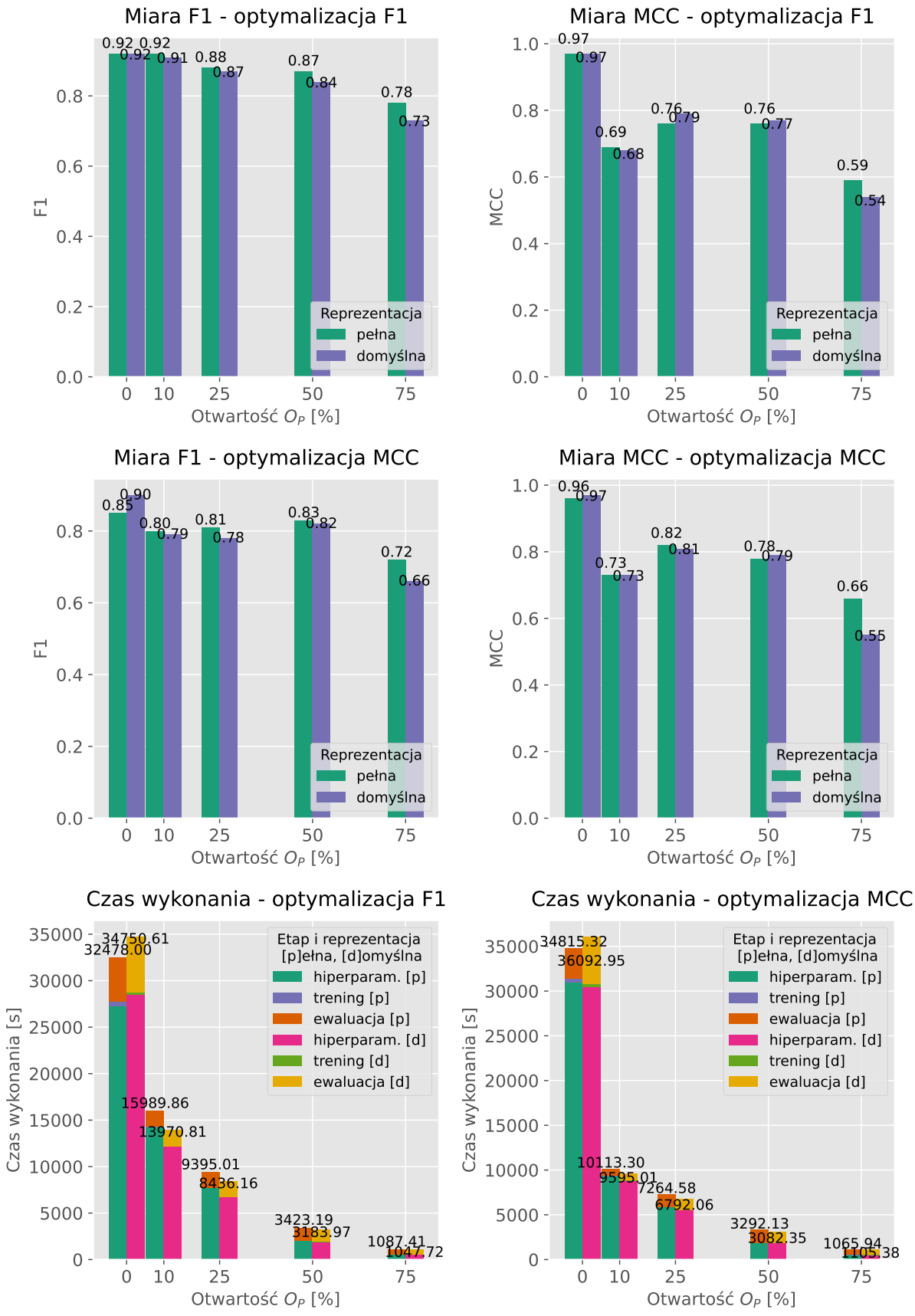
Rysunek 6.5: Zbiorcze zestawienie wyników klasyfikacji oraz czasów wykonania analizy algorytmu EVM dla pełnej i domyślnej reprezentacji żądań HTTP narzędzia *Hfinger*.

Wyniki klasyfikacji algorytmu EVM zaprezentowane na rys. 6.5 przedstawiają dosyć podobne poziomy miar F1 i MCC. W większości przypadków pełna reprezentacja jest lepsza od domyślnej o 0,01 lub 0,02 (co najwyżej 4%), albo są na takim samym poziomie. Jedynie w przypadku otwartości 0% i miary MCC dla wyboru hiperparametrów względem miary F1 reprezentacja domyślna jest lepsza od pełnej o 0,01 (1%). Podobnie się ma z czasami wykonania. Bez względu na miarę używaną do wyboru hiperparametrów, sumaryczny czas wykonania analizy jest dosyć podobny i pełna reprezentacja zajmuje co najwyżej ok. 3% czasu więcej niż domyślna reprezentacja.



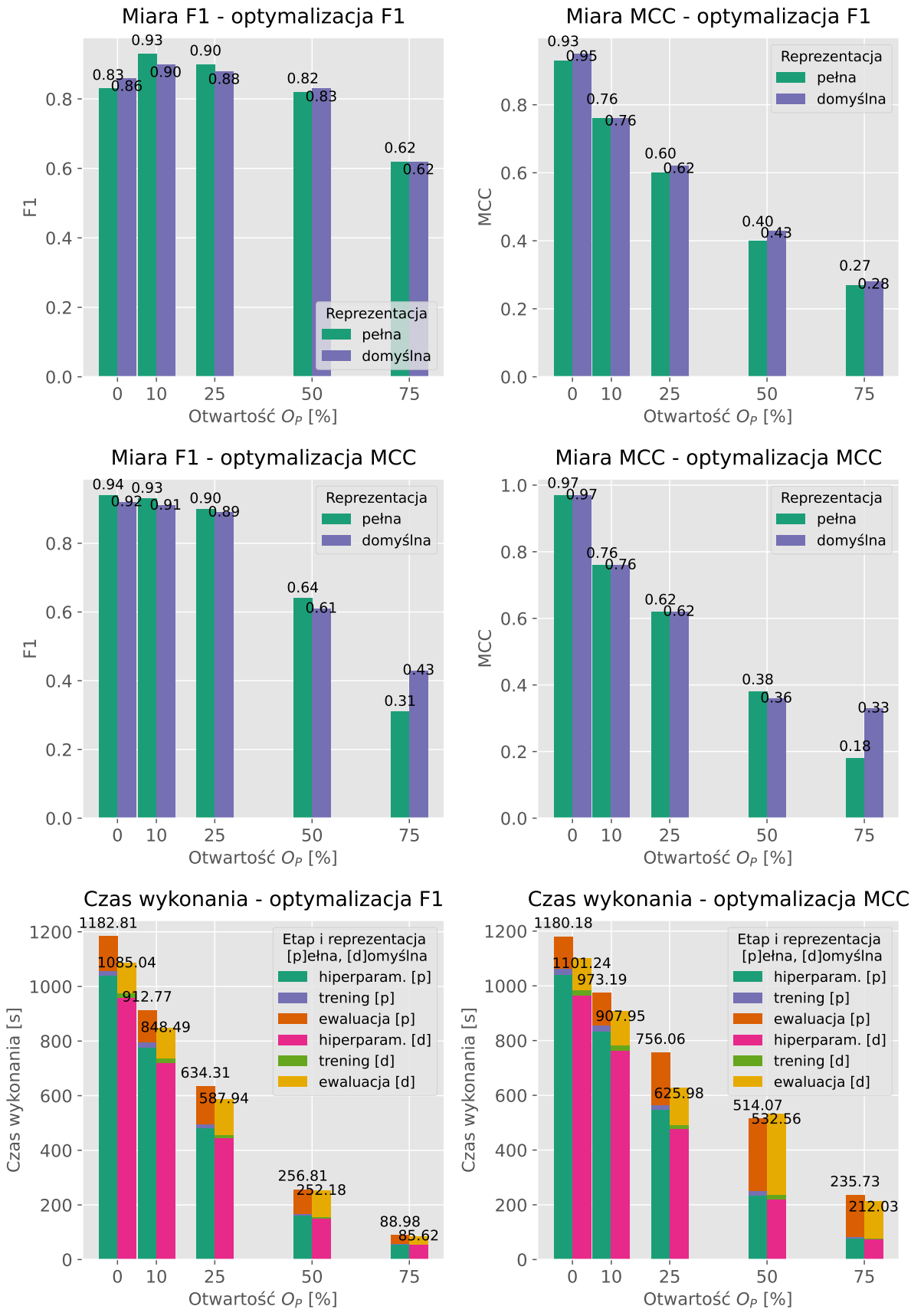
Rysunek 6.6: Zbiorcze zestawienie wyników klasyfikacji oraz czasów wykonania analizy algorytmu W-SVM dla pełnej i domyślnej reprezentacji żądań HTTP narzędzia *Hfinger*.

Na rys. 6.6 poziomy klasyfikacji algorytmu W-SVM różnią się znacząco między dwiema reprezentacjami. Przy wyborze hiperparametrów względem miary F1 poziomy miary F1 dla domyślnej reprezentacji zawierają się w zakresie od 0,54 dla $O_P = 75\%$ do 0,66 dla $O_P = 0\%$, co jest wynikiem mniejszym od poziomów pełnej reprezentacji o od ok. 17% do ok. 24%. Pozostając przy tej samej metodzie optymalizacji hiperparametrów, zauważalne są niższe wartości miary MCC domyślnej reprezentacji względem pełnej. Różnica poziomów zawiera się w zakresie od 0,01 do 0,1, co przekłada się na zmniejszenie poziomów miary od ok. 2% do 20%. Podobne zależności są widoczne dla scenariusza wyboru hiperparametrów względem miary MCC. Różnice poziomów miar między reprezentacjami nie są aż tak duże jak przy optymalizacji względem miary F1: wyniki klasyfikacji są mniejsze dla domyślnej reprezentacji w granicach 4-11% dla F1 i 1-11% dla MCC. Niemniej w przypadku $O_P = 50\%$ domyślna reprezentacja osiąga o 0,02 (5%) wyższy poziom miary F1 oraz o 0,06 (26%) wyższą wartość miary MCC niż pełna reprezentacja. Zależności te są odzwierciedlone w sumarycznym czasie wykonania analizy. Przy wyborze hiperparametrów względem miary F1 czas wykonania analizy przy domyślnej reprezentacji był o od 13% do 42% krótszy niż dla pełnej reprezentacji. Z kolei przy wyborze hiperparametrów względem miary MCC, czas wykonania analizy przy domyślnej reprezentacji był o od 5% do 43% krótszy niż dla pełnej reprezentacji.



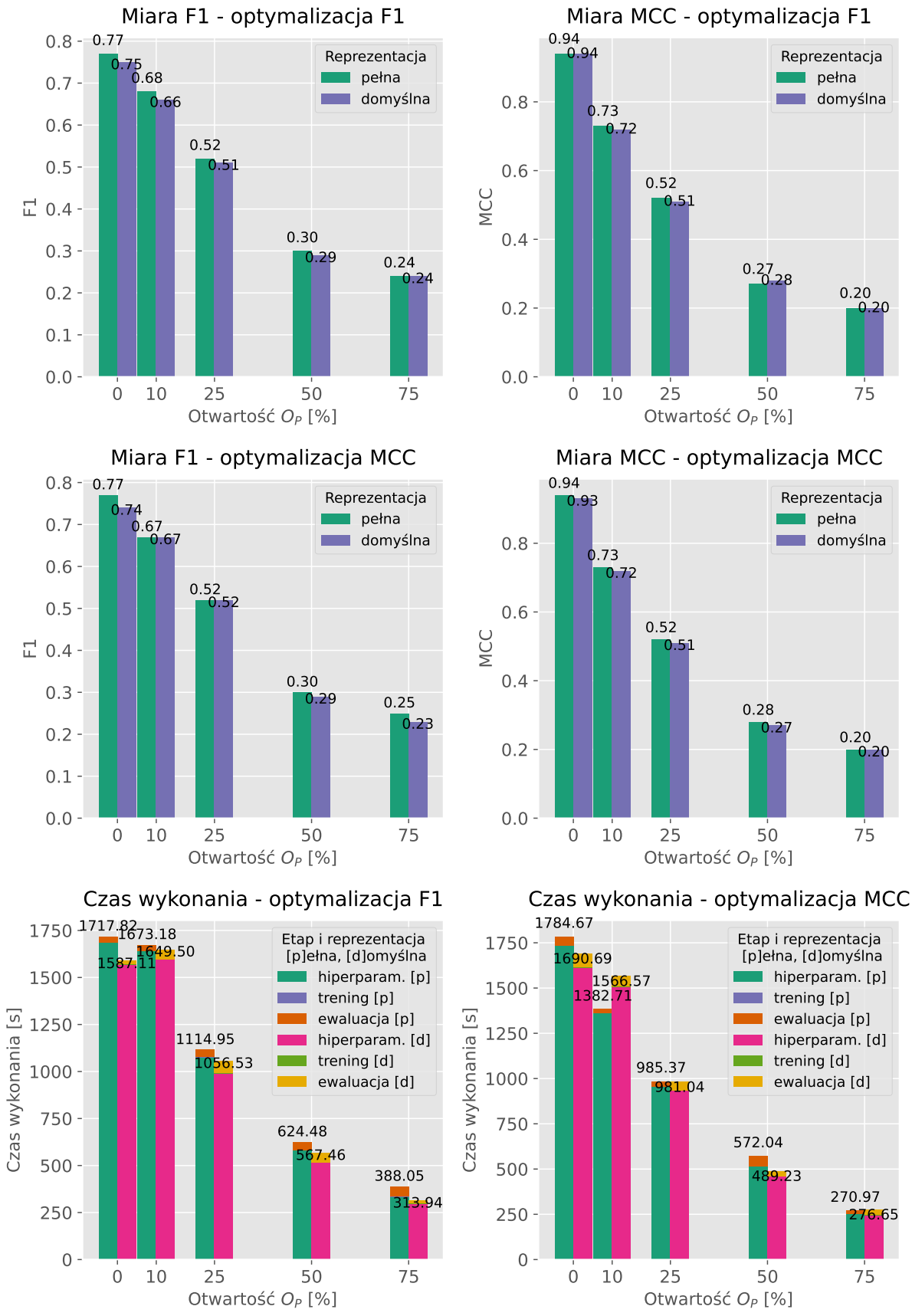
Rysunek 6.7: Zbiorcze zestawienie wyników klasyfikacji oraz czasów wykonania analizy algorytmu OSSVM dla pełnej i domyślnej reprezentacji żądań HTTP narzędzia *Hfinger*.

Porównanie poziomów klasyfikacji między pełną i domyślną reprezentacją żądań HTTP dla algorytmu OSSVM przedstawiono na rys. 6.7. Przy wyborze hiperparametrów względem miary F1 poziomy miary F1 są równe dla obu reprezentacji przy scenariuszu zamkniętozbiorowym, jednak wraz ze wzrostem otwartości wzrasta także różnica między nimi na niekorzyść domyślnej reprezentacji. Różnica ta waha się od 1% do 6%. Pozostając przy tym samym scenariuszu wyboru hiperparametrów, wartość miary MCC jest równa dla obu scenariuszy przy zerowej otwartości. Dla $O_P = 10\%$ oraz $O_P = 75\%$ domyślna reprezentacja osiąga odpowiednio o 1% i 8% mniejsze poziomy miary MCC. Natomiast dla $O_P = 25\%$ i $O_P = 50\%$ osiąga odpowiednio o 4% i o 1% większe wartości niż pełna reprezentacja. Przy wyborze hiperparametrów względem miary MCC dla scenariusza zamkniętozbiorowego domyślna reprezentacja uzyskała wyższy o 6% poziom miary F1 i o 1% wyższy poziom miary MCC. Dla innych poziomów otwartości domyślna reprezentacja miała poziomy miary F1 niższe o od 1% do 8%. Z kolei dla miary MCC przy $O_P = 10\%$ jej wartość była taka sama dla obydwu reprezentacji, a dla $O_P = 50\%$ większa o 1% dla domyślnej reprezentacji. Przy otwartości $O_P = 25\%$ pełna reprezentacja była lepsza o 1%, natomiast dla $O_P = 75\%$ o 20%. Sumaryczne czasy wykonania analizy wskazują na wydłużenie czasu obliczeń domyślnej reprezentacji dla otwartości $O_P = 0\%$ o 7% i 4% dla wyboru hiperparametrów względem miary odpowiednio F1 i MCC. Dla innych wartości otwartości przy wyborze względem miary F1 czasy wykonania dla domyślnej reprezentacji są krótsze od pełnej reprezentacji w zakresie od 4% do 13%. Przy wyborze hiperparametrów względem miary MCC dla poziomów otwartości w zakresie 10-50% domyślna reprezentacja osiąga wartości niższe w zakresie od 5% do 7%. Natomiast dla otwartości $O_P = 75\%$ czas wykonania dla domyślnej reprezentacji jest wyższy o 4%.



Rysunek 6.8: Zbiorcze zestawienie wyników klasyfikacji oraz czasów wykonania analizy algorytmu MCR-SVM dla pełnej i domyślnej reprezentacji żądań HTTP narzędzia *Hfinger*.

Wartości poziomów klasyfikacji algorytmu MCR-SVM dla pełnej i domyślnej reprezentacji żądań zaprezentowane na rys. 6.8 wskazują na pewne podobieństwo wyników. Analizując wartość miary F1 przy wyborze hiperparametrów względem właśnie tej miary, można dostrzec o 4% wyższy poziom dla domyślnej reprezentacji w przypadku $O_P = 0\%$ oraz o 1% wyższy dla $O_P = 50\%$. Dla innych otwartości wartości miary F1 są takie same ($O_P = 75\%$) lub pełna reprezentacja osiąga wyniki wyższe o 3% ($O_P = 10\%$) lub 2% ($O_P = 25\%$). Wartości miary MCC przy tym samym scenariuszu optymalizacji hiperparametrów są wyższe dla domyślnej reprezentacji w czterech przypadkach: o 2% dla $O_P = 0\%$, 3% dla $O_P = 25\%$, 8% dla $O_P = 50\%$ oraz 4% dla $O_P = 75\%$. Dla otwartości $O_P = 10\%$ są równe. Przy wyborze hiperparametrów względem miary MCC, pełna reprezentacja osiąga wyższy poziom miary F1 w stosunku do domyślnej reprezentacji w zakresie od 1% do 5%. Choć dla $O_P = 75\%$ sytuacja się odwraca i domyślna reprezentacja osiąga wynik o 39% wyższy od pełnej reprezentacji. Pozostając przy tym samym scenariuszu optymalizacji hiperparametrów, obydwie reprezentacje osiągają takie same wartości miary MCC dla otwartości od 0% aż do 25%. Dla $O_P = 50\%$ pełna reprezentacja jest lepsza o 6%, ale dla $O_P = 75\%$ domyślna reprezentacja osiąga poziom o 78% wyższy. Sumaryczne czasy wykonania analizy przy wyborze hiperparametrów względem miary F1 są niższe dla domyślnej reprezentacji w zakresie od 2% do 8%. Natomiast przy wyborze hiperparametrów względem miary MCC domyślna reprezentacja osiąga czasy niższe o od 1% do 17%. Jednak dla $O_P = 50\%$ użycie domyślnej reprezentacji spowodowało wzrost czasu ewaluacji, przez co sumaryczny czas analizy był wyższy o 4% niż dla pełnej reprezentacji.



Rysunek 6.9: Zbiorcze zestawienie wyników klasyfikacji oraz czasów wykonania analizy algorytmu RF dla pełnej i domyślnej reprezentacji żądań HTTP narzędzia *Hfinger*.

Porównanie wyników klasyfikacji algorytmu RF wykorzystującego reprezentację pełną i domyślną umieszczone na rys. 6.9 wskazuje na ich podobieństwo, choć w ograniczonym stopniu. Przy wyborze hiperparametrów względem miary F1, dla miary F1, domyślna reprezentacja osiąga niższe wyniki o od 2% do 3%. Jednak dla otwartości $O_P = 75\%$ domyślna reprezentacja osiąga taki sam poziom miary F1 jak pełna reprezentacja. Pozostając przy tym samym scenariuszu optymalizacji hiperparametrów, poziomy miary MCC są równe dla obydwu reprezentacji dla $O_P = 0\%$ i $O_P = 75\%$. Dla $O_P = 10\%$ i $O_P = 25\%$ wartości MCC są niższe dla domyślnej reprezentacji o, odpowiednio, 1% i 2%. Przy $O_P = 50\%$ domyślna reprezentacja osiąga o 4% większy wynik. W scenariuszu optymalizacji hiperparametrów względem miary MCC domyślna reprezentacja osiąga poziomy miary F1 mniejsze w zakresie od 3% do 8% lub takie same jak pełna reprezentacja. Dla miary MCC domyślna reprezentacja osiąga poziomy o od 1% do 4% niższe niż pełna reprezentacja, choć dla $O_P = 75\%$ obie reprezentacje osiągają ten sam wynik. Porównanie sumarycznych czasów analizy w przypadku scenariusza wyboru hiperparametrów względem miary F1 wskazuje, że domyślna reprezentacja zajmuje od 1% do 19% mniej czasu niż pełna reprezentacja. Natomiast dla scenariusza wyboru hiperparametrów względem miary MCC sumaryczne czasy wykonania analizy dla dwóch przypadków $O_P = 10\%$ i $O_P = 75\%$ są większe dla domyślnej reprezentacji niż dla pełnej o odpowiednio 13% i 2%. W pozostałych przypadkach pełna reprezentacja osiąga czasy większe co najwyżej o 17%.

Przedstawione powyżej porównanie wykazało, że dla wszystkich badanych algorytmów domyślna reprezentacja żądań HTTP tworzona przez narzędzie *Hfinger* w większości przypadków ma takie same lub niższe poziomy klasyfikacji oraz sumaryczne czasy wykonania analizy w stosunku do pełnej reprezentacji. Różnice poziomów klasyfikacji to najczęściej zakres 0,01-0,04, choć dla algorytmu W-SVM i pewnych przypadków OSSVM (np. $O_P = 75\%$) różnice były większe. Czasami dochodziło do przypadków, gdy domyślna reprezentacja osiągała wyższe poziomy klasyfikacji niż pełna reprezentacja, jak np. dla MCR-SVM dla $O_P = 0\%$ i $O_P = 50\%$ (wybór hiperparametrów względem F1), ale różnica ta nie była zachowana dla innych otwartości.

Opisane zmiany w wartościach poziomów klasyfikacji między przedstawionymi powyżej dwoma reprezentacjami żądań nie były na tyle znaczące, aby wpłynąć na ocenę samych algorytmów. Zachowane zostały zależności, które wskazywały na dobre wyniki detekcji algorytmów OSSVM i MCR-SVM w stosunku do pozostałych, a także na dobre przełożenie poziomu detekcji do czasu wykonania analizy algorytmu EVM.

Powyższa analiza porównawcza udowodniła, że domyślna reprezentacja żądań HTTP narzę-

dzia *Hfinger* umożliwia uzyskanie dobrych wyników klasyfikacji, które w większości przypadków nie odbiegają wartościami od tych uzyskanych przy wykorzystaniu pełnej reprezentacji. Ograniczona liczba cech oraz typ zaokrąglenia tych wykorzystanych w analizowanej reprezentacji są wystarczające do dokonania klasyfikacji na dobrym poziomie efektywności. Z tego powodu można uznać, że wybór cech tworzących domyślną reprezentację, dokonany przy tworzeniu narzędzia Hfinger, został wykonany poprawnie.

Pomimo dobrych wyników klasyfikacji domyślnej reprezentacji żądań HTTP narzędzia *Hfinger*, w kolejnej części badań wykorzystana zostanie pełna reprezentacja, która jednak w większości przypadków uzyskuje wyższe wartości badanych miar klasyfikacji.

6.5.3 Porównanie wydajności reprezentacji narzędzia *Hfinger* oraz reprezentacji trigramowej

W ostatnim etapie badań porównano dwie reprezentacje żądań HTTP: pełną reprezentację utworzoną przez narzędzie *Hfinger* oraz reprezentację trigramową. Ma to na celu ocenę przydatności narzędzia *Hfinger* do klasyfikacji złośliwego oprogramowania na tle innego sposobu reprezentacji żądań. Aby przeprowadzić analizę eksperymentalną, użyto dwóch algorytmów uczenia maszynowego wybranych na wcześniejszym etapie analizy tj. EVM oraz MCR-SVM. Algorytmy te zostały wybrane ze względu na dobry stosunek wyników klasyfikacji do czasu wykonania analizy (por. rozdział 6.5.1). Dodatkowym wymiarem analizy jest fakt, że EVM jest algorytmem zaprojektowanym do scenariusza otwartozbiorowego, natomiast MCR-SVM bazuje na standardowym, zamkniętozbiorowym algorytmie SVM, rozszerzonym o mechanizm odrzucania niepewnych klasyfikacji w celu wykrycia klas nieznanych.

Oprócz wyników klasyfikacji zbadane zostały czasy wykonania analizy. Na wyniki znaczący wpływ mają zasoby sprzętowe użyte do badań. Należy przypomnieć, że badania reprezentacji trigramowej zostały wykonane na innej maszynie niż pełna reprezentacja narzędzia *Hfinger*. Było to spowodowane dużym zużyciem pamięci operacyjnej koniecznym do obsługi reprezentacji trigramowej. Ze względu na dużą liczbę cech (wymiarowość) oraz liczbę rekordów danych przechowujących reprezentację żądań, ilość przetwarzanych danych, pomimo optymalizacji tego procesu, była duża. Mimo dostępnych 64GB pamięci RAM, w badaniach można było użyć co najwyżej 8 wątków obliczeniowych, ponieważ dane konieczne do uruchomienia obliczeń w kolejnych wątkach nie mieściły się w pamięci. Zatem przy wyborze hiperparametrów względem miary F1 dla otwartości od 0% do 10% używano 6 wątków obliczeniowych, a od 25% i więcej 8.

W przypadku wyboru względem miary MCC granica użycia 8 wątków obliczeniowych została przesunięta z poziomu otwartości 25% do 50%. Odnosi się to obu wybranych algorytmów. Powyższe wyjaśnienie jest ważne z dwóch powodów. Po pierwsze umożliwia właściwe porównanie czasów analizy między reprezentacjami i poprawną interpretację uzyskanych wyników. Po drugie ujawnia dość dużą wadę reprezentacji trigramowej, która wymaga zdecydowanie większych zasobów pamięci RAM w stosunku do reprezentacji bazującej na narzędziu *Hfinger*.

Wyniki klasyfikacji algorytmów EVM oraz MCR-SVM wykorzystujących reprezentację trigramową dla dwóch scenariuszy wyboru hiperparametrów znajdują się w tab. 6.9.

Algorytm	Optymalizacja F1				Optymalizacja MCC			
	EVM		MCR-SVM		EVM		MCR-SVM	
Otwartość O_P [%]	F1	MCC	F1	MCC	F1	MCC	F1	MCC
0	0,55	0,50	0,56	0,61	0,57	0,50	0,53	0,59
10	0,53	0,41	0,53	0,45	0,53	0,41	0,52	0,44
25	0,64	0,46	0,61	0,42	0,64	0,46	0,53	0,40
50	0,85	0,73	0,74	0,30	0,85	0,76	0,71	0,37
75	0,78	0,59	0,58	0,23	0,78	0,60	0,57	0,36

Tabela 6.9: Efektywność klasyfikacji badanych algorytmów dla reprezentacji trigramowej żądań HTTP dla wyboru hiperparametrów względem miary F1 i MCC.

Wyniki w tab. 6.9 wskazują na podobieństwo poziomów miary F1 dla $O_P = 0\%$ i $O_P = 10\%$ dla obu algorytmów przy wyborze hiperparametrów względem miary F1, choć dla $O_P = 0\%$ MCR-SVM jest lepszy o 2%. Niemniej odpowiadające tym poziomom otwartości wartości MCC są wyższe dla MCR-SVM o odpowiednio 22% i 10%. Sytuacja zmienia się wraz ze wzrostem otwartości od $O_P = 25\%$. Algorytm EVM osiąga lepsze wyniki od 5% do 34% pod względem miary F1 oraz od 10% do 157% wyższe poziomy miary MCC. Przy wyborze hiperparametrów względem miary MCC widać podobną zależność jak w poprzednim scenariuszu wyboru hiperparametrów: dla $O_P = 0\%$ i $O_P = 10\%$ wartości miary MCC są wyższe dla algorytmu MCR-SVM o odpowiednio 18% i 7%, choć miara F1 wskazuje na lepsze wyniki algorytmu EVM o 8% i 2%. Ze wzrostem otwartości od $O_P = 25\%$ wyniki EVM są znacząco lepsze pod względem obydwu miar: od 20% do 37% dla F1 oraz od 15% do 105% dla MCC.

W tab. 6.10 zawarto czasy wykonania trzech etapów analizy dla algorytmów EVM i MCR-SVM.

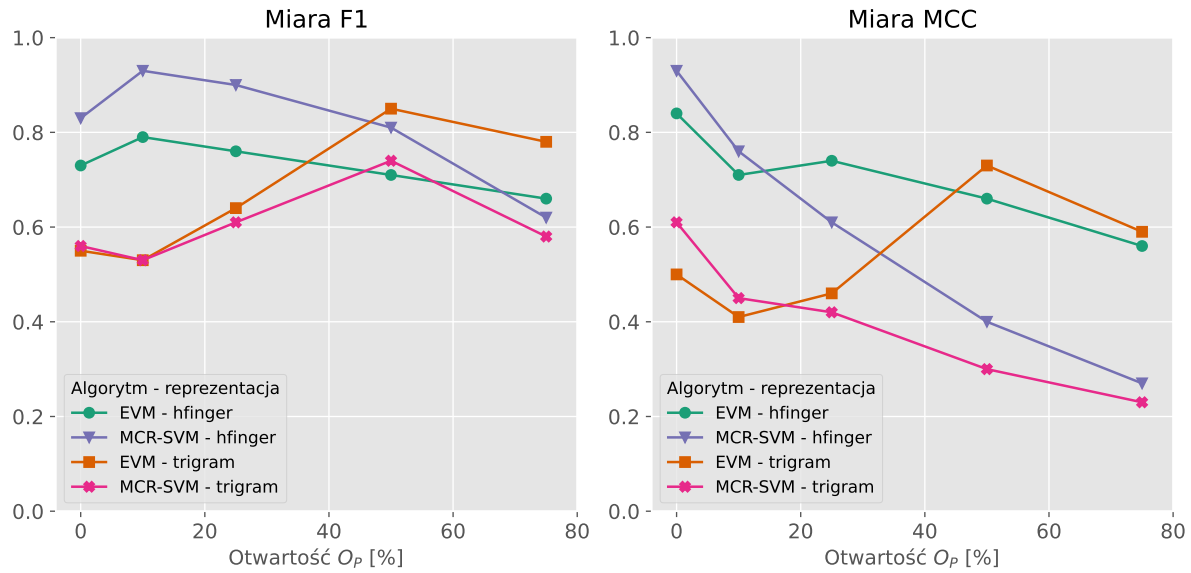
		Optymalizacja F1			Optymalizacja MCC		
Algorytm	O_P [%]	Hiper. [s]	Trening [s]	Ewaluacja [s]	Hiper. [s]	Trening [s]	Ewaluacja [s]
EVM	0	14912,12	55,36	1092,95	17602,37	53,90	1369,67
	10	12181,26	44,80	684,80	15904,97	45,89	1734,58
	25	8156,41	31,91	1530,67	10586,47	31,47	1827,65
	50	3013,05	17,38	1153,60	3096,92	17,13	1228,85
	75	1685,62	6,86	647,56	1740,98	6,72	868,44
MCR-SVM	0	24425,73	329,61	1774,24	27432,83	413,81	1993,22
	10	21354,16	274,13	1994,65	23921,01	302,53	2021,14
	25	10070,44	149,85	2237,11	14149,92	241,53	3302,65
	50	2994,00	48,99	1115,04	4770,97	141,96	2450,45
	75	946,61	12,99	501,25	1367,00	36,02	1330,50

Tabela 6.10: Czasy wykonania trzech etapów analizy przez badane algorytmy dla reprezentacji trigramowej żądań HTTP (nazwa etapu wyboru hiperparametrów została skrócona do słowa *hiper.* dla zaoszczędzenia miejsca w tabeli).

Zsumowanie czasów analizy z tab. 6.10 wskazuje, że algorytm MCR-SVM potrzebuje więcej czasu na obliczenia niż EVM o od 28% do 83% przy wyborze hiperparametrów względem miary F1, choć dla $O_P = 50\%$ EVM potrzebuje 1% więcej czasu, a dla $O_P = 75\%$ nawet 60% czasu. W przypadku wyboru hiperparametrów względem miary MCC, EVM wykonuje sumaryczne obliczenia od 4% do 70% szybciej niż MCR-SVM. Widoczne jest także wydłużenie sumarycznych czasów wykonania obliczeń między scenariuszami wyboru hiperparametrów. Dla obu algorytmów dla miary MCC są one większe w zakresie od 4% do 37% dla EVM, natomiast dla MCR-SVM w zakresie od 11% do 87%.

Porównanie wyników klasyfikacji algorytmów EVM i MCR-SVM w scenariuszu wyboru hiperparametrów względem miary F1 dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger* i reprezentacji trigramowej zilustrowano na rys. 6.10. Na rysunku tym widać, że dla miary F1 aż do $O_P = 25\%$ pełna reprezentacja narzędzia *Hfinger* osiąga wyższe wartości niż reprezentacja trigramowa: dla EVM od 18% do 48%, a dla MCR-SVM od 46% do 74%. Sytuacja zmienia się dla otwartości na poziomie 50%. Następuje wzrost wartości miary F1 obydwu algorytmów przy korzystaniu z reprezentacji trigramowej, jednak tylko EVM osiąga wartości wyższe o 16% dla obydwu przypadków $O_P = 50\%$ i $O_P = 75\%$ niż dla pełnej reprezentacji *Hfinger*. Dla algorytmu MCR-SVM wartości miary F1 przy wykorzystaniu pełnej reprezentacji *Hfinger* są nadal wyższe o 11% i 7% dla odpowiednio $O_P = 50\%$ i $O_P = 75\%$. Dla algorytmu EVM w przypadku miary

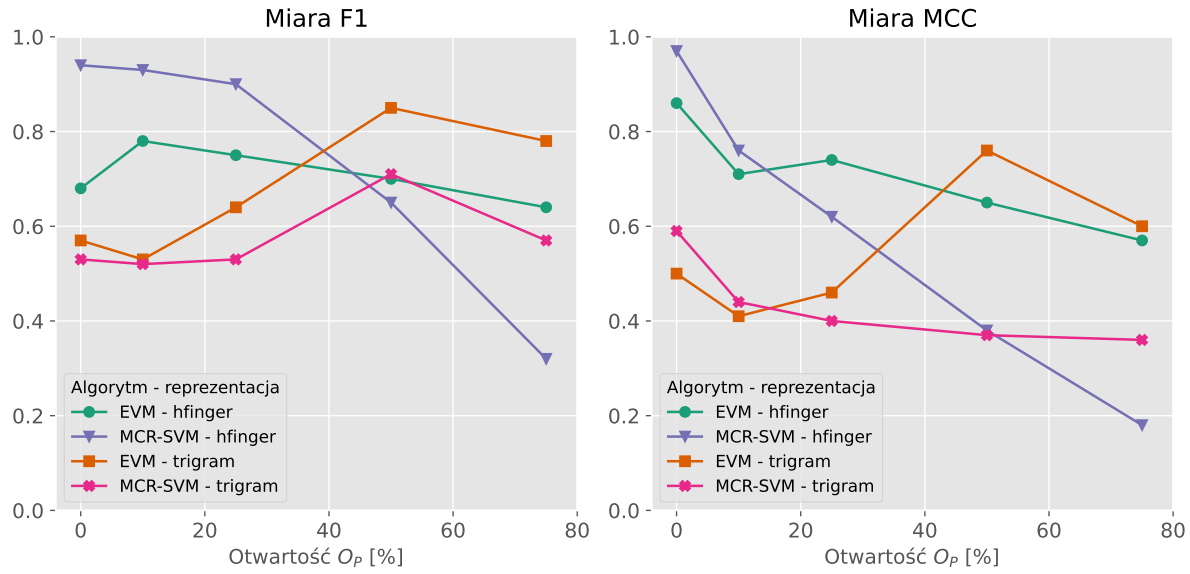
MCC dochodzi do podobnej sytuacji jak dla miary F1. Wartości MCC aż do $O_P = 25\%$ są wyższe dla pełnej reprezentacji *Hfinger* od 61% do 72%. Jednak ze wzrostem otwartości dla $O_P = 50\%$ i $O_P = 75\%$ reprezentacja trigramowa osiąga lepsze wyniki o odpowiednio 10% i 5%. Natomiast algorytm MCR-SVM używający pełnej reprezentacji *Hfinger* dla wszystkich poziomów otwartości osiągnął wyższe wartości miary MCC niż reprezentacja trigramowa w zakresie od 18% do 67%.



Rysunek 6.10: Porównanie wyników klasyfikacji dwóch badanych algorytmów dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger* oraz reprezentacji trigramowej (wybór hiperparametrów względem miary F1).

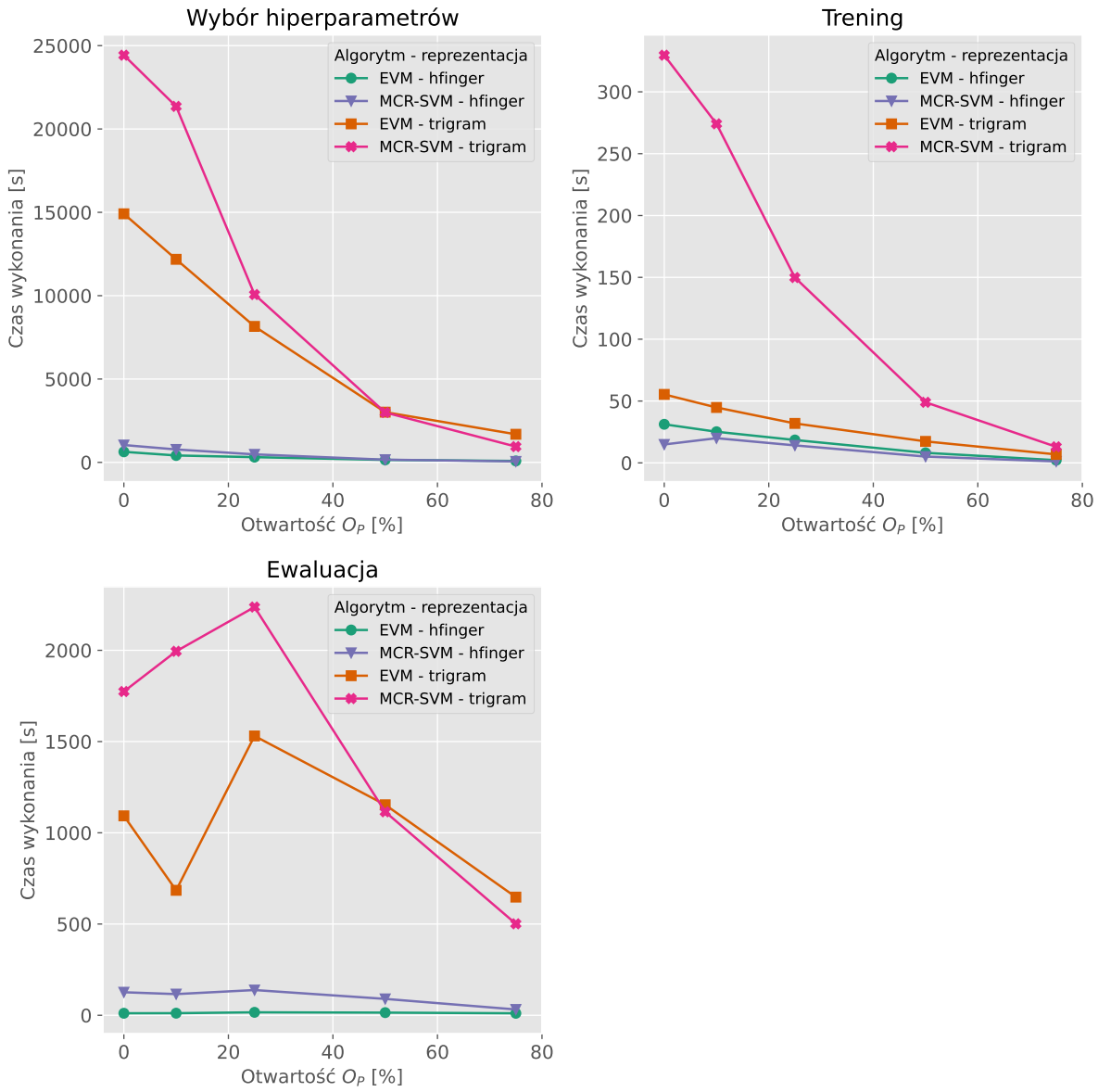
Porównanie wyników klasyfikacji algorytmów EVM i MCR-SVM w scenariuszu wyboru hiperparametrów względem miary MCC dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger* i reprezentacji trigramowej przedstawiono na rys. 6.11. Na rysunku tym można zauważyć, że aż do $O_P = 25\%$ wartości miary F1 dla obydwu algorytmów są większe dla pełnej reprezentacji *Hfinger* od reprezentacji trigramowej: dla EVM od 17% do 45%, a dla MCR-SVM od 70% do 78%. Dla otwartości $O_P = 50\%$ i $O_P = 75\%$ dochodzi do zamiany i reprezentacja trigramowa osiąga lepsze wyniki: o 21% wyższe dla obydwu otwartości w przypadku algorytmu EVM oraz o odpowiednio 12% i 80% dla algorytmu MCR-SVM. Wyniki miary MCC algorytmu EVM również przedstawiają podobną zależność. Do otwartości $O_P = 25\%$ pełna reprezentacja *Hfinger* osiąga wartości wyższe o od 59% do 73%, natomiast dla $O_P = 50\%$ i $O_P = 75\%$ to reprezentacja trigramowa jest lepsza o odpowiednio 17% i 6%. Dla algorytmu MCR-SVM

pełna reprezentacja *Hfinger* jest lepsza o od 3% do 68% aż do $O_P = 50\%$, ale dla $O_P = 75\%$ to reprezentacja trigramowa osiąga lepszy o 96% wynik.



Rysunek 6.11: Porównanie wyników klasyfikacji dwóch badanych algorytmów dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger* oraz reprezentacji trigramowej (wybór hiperparametrów względem miary MCC).

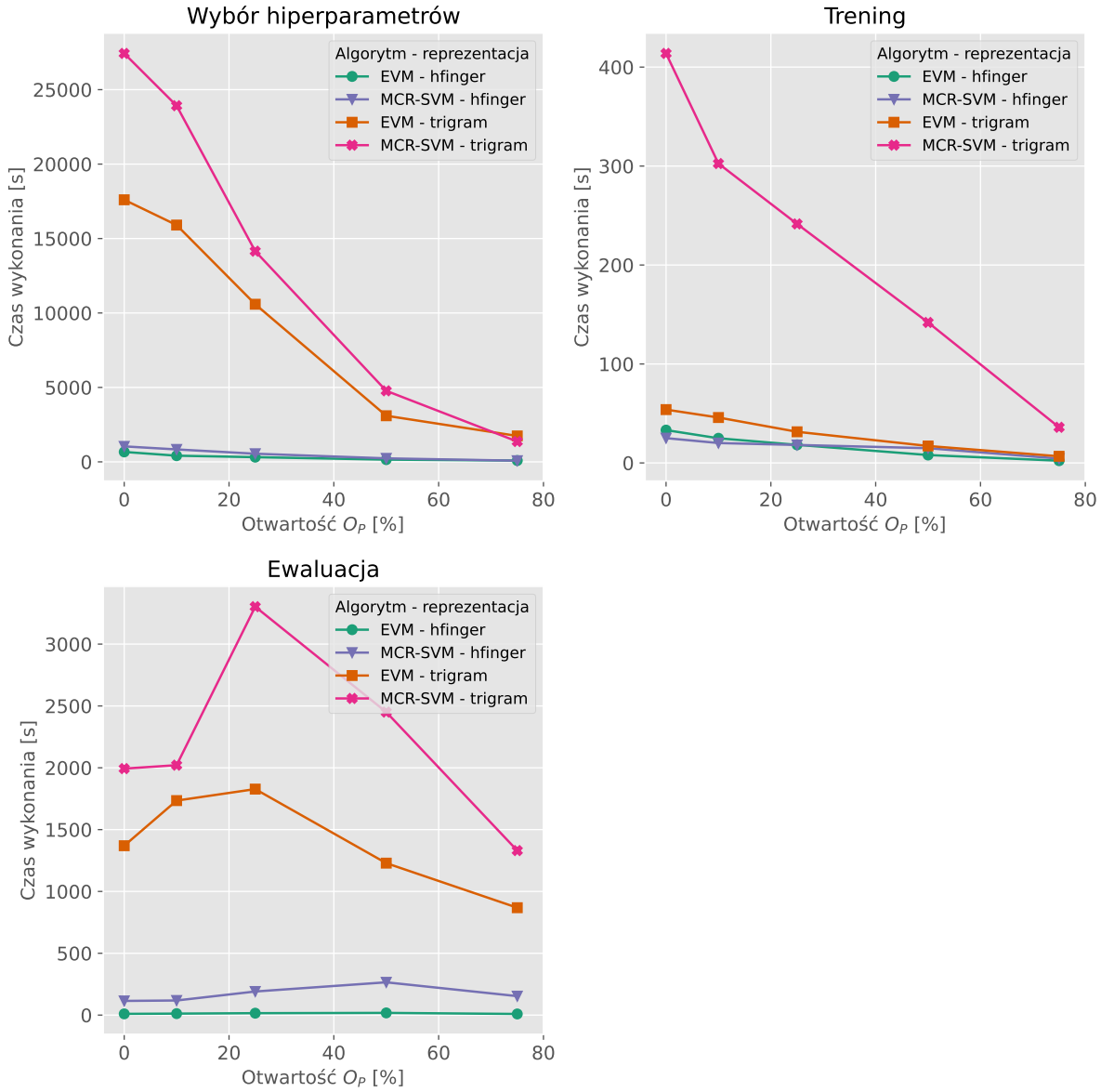
Na rys. 6.12 znajduje się porównanie czasów trzech etapów analizy dla algorytmów EVM i MCR-SVM dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger* oraz reprezentacji trigramowej. Rysunek ten dotyczy wyboru hiperparametrów względem miary F1. Różnica czasu wykonania etapu wyboru hiperparametrów między reprezentacją trigramową a pełną reprezentacją *Hfinger* jest znacząca. W przypadku algorytmu EVM reprezentacja trigramowa zajmowała od 20 do 29 razy więcej czasu, natomiast dla algorytmu MCR-SVM od 17 do 28 razy. Różnica czasów między reprezentacjami w fazie treningu była dla algorytmu EVM mniejsza niż dla MCR-SVM. Reprezentacja trigramowa zajmowała jedynie o od 72% do 214% dłużej. Natomiast dla MCR-SVM i reprezentacji trigramowej czasy treningu były od 10 do 22 razy dłuższe. Czasy ewaluacji algorytmu EVM były dla reprezentacji trigramowej od 60 do 99 razy większe niż dla pełnej reprezentacji *Hfinger*. W przypadku algorytmu MCR-SVM także dla reprezentacji trigramowej czasy te były od 13 do 17 razy większe.



Rysunek 6.12: Porównanie czasów wykonania faz analizy przez dwa badane algorytmy dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger* oraz reprezentacji trigramowej (wybór hiperparametrów względem miary F1).

Rys. 6.13 przedstawia porównanie czasów trzech etapów analizy dla algorytmów EVM i MCR-SVM dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger* oraz reprezentacji trigramowej. Dotyczy on scenariusza wyboru hiperparametrów względem miary MCC. Etap wyboru hiperparametrów dla algorytmu EVM i reprezentacji trigramowej zajmował od 20 do 38 razy więcej czasu niż dla tego samego algorytmu wykorzystującego pełną reprezentację *Hfinger*. Z kolei dla algorytmu MCR-SVM z reprezentacją trigramową zajmowało to od 18 do 29 razy więcej czasu. Etap treningu dla algorytmu EVM z reprezentacją trigramową trwał o od 62%

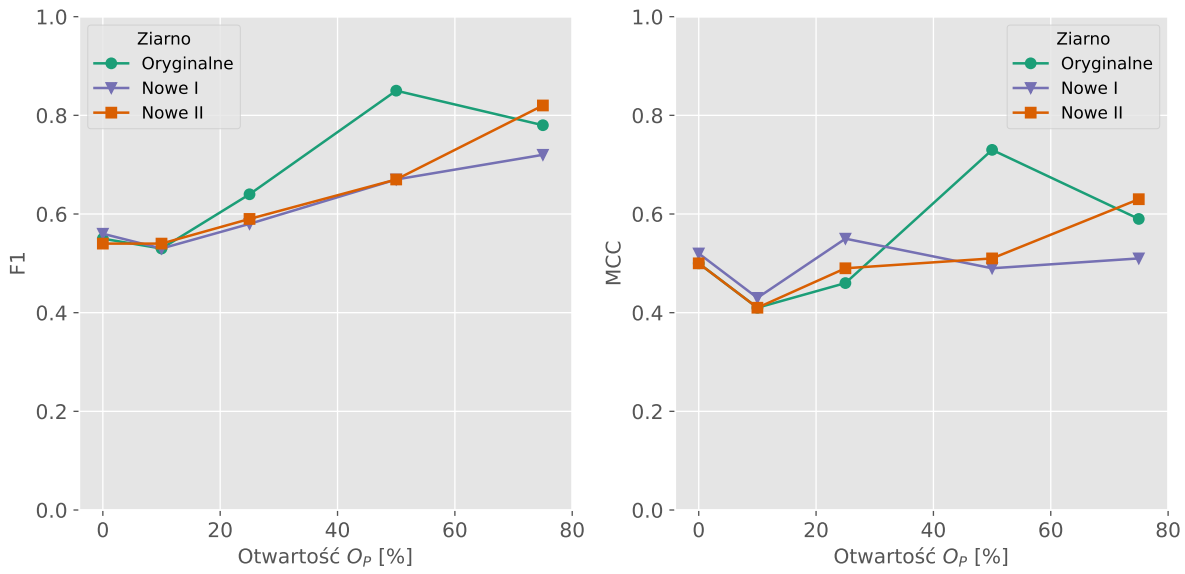
do 198% dłużej. Różnica ta jest mniejsza niż dla algorytmu MCR-SVM, któremu zajmowało to od 8 do 16 razy dłużej, gdy wykorzystywał reprezentację trigramową. Do bardzo dużej różnicy w czasach dla algorytmu EVM doszło na etapie ewaluacji. Wykorzystanie reprezentacji trigramowej spowodowało wydłużenie czasu od 70 do 145 razy. Dla algorytmu MCR-SVM i reprezentacji trigramowej czasy te wydłużyły się od 9 do 17 razy.



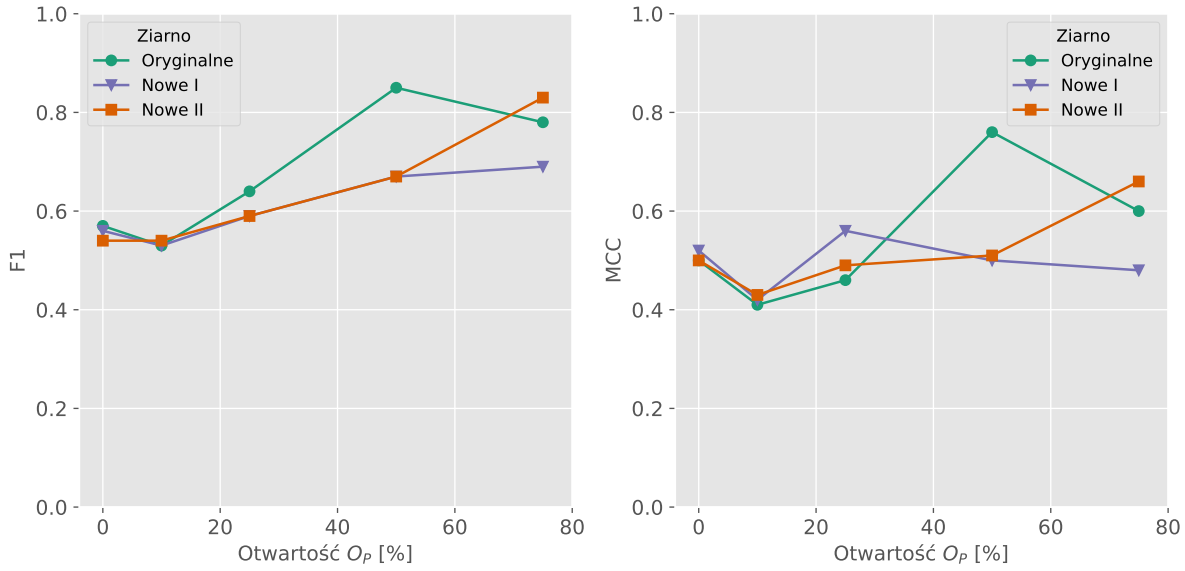
Rysunek 6.13: Porównanie czasów wykonania faz analizy przez dwa badane algorytmy dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger* oraz reprezentacji trigramowej (wybór hiperparametrów względem miary MCC).

Na rys. 6.10 i 6.11 widoczny jest duży wzrost wartości F1 dla otwartości powyżej $O_P = 50\%$ dla obu algorytmów wykorzystujących reprezentację trigramową. Taki znaczący wzrost jest

widoczny również dla wartości miary MCC, ale tylko dla algorytmu EVM. Dla pełnej reprezentacji *Hfinger* wartości miar klasyfikacji zmniejszają się wraz ze wzrostem otwartości, co można uzasadnić zwiększaniem się liczby klas nieznanych, co stawia większe wymogi przed możliwościami klasyfikacyjnymi i wykrywaniem tych klas przez używane algorytmy. Natomiast wzrosty miar klasyfikacyjnych przy wzroście otwartości widoczne dla reprezentacji trigramowej przedstawiają odwrotną zależność. Dodatkowo zależność ta jest widoczna dla obu algorytmów, co potencjalnie wskazywałoby na częściową niezależność od wykorzystywanego algorytmu. By wykluczyć wpływ mechanizmu wyboru klas i próbek danych cały cykl badań dla algorytmu EVM i reprezentacji trigramowej powtórzono dwukrotnie. Wykorzystano do tego dwie dodatkowe, różne wartości ziarna generatora liczb pseudolosowych. Uzyskane wyniki przedstawiono na rys. 6.14 dla wyboru hiperparametrów względem miary F1 oraz rys. 6.15 dla wyboru hiperparametrów względem miary MCC. Na obu rysunkach widoczne są wzrosty miary F1 wraz ze wzrostem otwartości, choć dla $O_P = 50\%$ nie są one tak duże jak dla oryginalnego ziarna. Poziomy wartości MCC w badaniach z nowymi ziarnami nie wskazują jednoznacznie na wzrost wartości tej miary wraz ze wzrostem otwartości. Poziomy MCC dla ziarna oznaczonego jako *Nowe I* ze wzrostem otwartości nie osiągają takich wzrostów jak dla pozostałych ziaren. Niestety, ze względu na duże wymagania obliczeniowe i czasowe badań z wykorzystaniem reprezentacji trigramowej, nie można było wykonać większej liczby powtórzeń analiz z innymi wartościami ziarna generatora liczb pseudolosowych. Może to być celem przyszłych analiz. Niemniej jednak obecnie uzyskane wyniki wskazują, że wzrost wydajności klasyfikacji z wykorzystaniem reprezentacji trigramowej nie jest silnie powiązany z konkretnym wyborem zestawu danych. Choć widoczny jest wpływ wartości ziarna generatora liczb pseudolosowych na konkretne wartości wyników (por. P. Madhyastha i in. [71], gdzie wpływ ten jest badany dla sieci neuronowych).



Rysunek 6.14: Porównanie efektywności klasyfikacji algorytmu EVM używającego reprezentacji trigramowej dla oryginalnego ziarna generatora liczb pseudolosowych oraz dwóch nowych (wybór hiperparametrów względem miary F1).



Rysunek 6.15: Porównanie efektywności klasyfikacji algorytmu EVM używającego reprezentacji trigramowej dla oryginalnego ziarna generatora liczb pseudolosowych oraz dwóch nowych (wybór hiperparametrów względem miary MCC).

Z przedstawionego powyżej porównania widoczne jest, że dla poziomów otwartości do 25% pełna reprezentacja narzędzia *Hfinger* osiąga lepsze poziomy klasyfikacji niż reprezentacja

trigramowa. Dodatkowo, z dwóch algorytmów badanych w tej części, to MCR-SVM osiąga lepsze wyniki. Jednak dla otwartości od $O_P = 50\%$ wzwyż sytuacja się odwraca. Miary F1 i MCC wzrastają dla algorytmu EVM używającego reprezentacji trigramowej i ostatecznie osiąga on lepsze wyniki niż przy użyciu pełnej reprezentacji *Hfinger*, a także lepsze wyniki niż MCR-SVM bez względu na reprezentację użytą przez ten ostatni.

Takie zróżnicowanie poziomów klasyfikacji wskazywałoby na szansę uzyskania dobrych efektów przez użycie uczenia zespołowego (ensemble learning), w którym można by połączyć wyniki predykcji klasyfikatorów używających różnych reprezentacji żądań HTTP. Niestety należałoby się nad tym pomysłem mocno zastanowić ze względu na znaczący wzrost czasów analizy spowodowanych użyciem reprezentacji trigramowej. Przetwarzanie dużych ilości danych wymaganych przy analizie n-gramowej powoduje duże zapotrzebowanie na pamięć operacyjną oraz zasoby obliczeniowe, co należy wziąć pod rozwagę przy tworzeniu systemu klasyfikacji bazującego na takiej reprezentacji.

6.6 Podsumowanie

Przedstawione powyżej analizy umożliwiły wypełnienie głównych celów badawczych: przeprowadzenia porównania wydajności klasyfikacji otwartozbiorowej złośliwego oprogramowania różnych algorytmów uczenia maszynowego, a także porównania wydajności tej klasyfikacji przy zastosowaniu różnych reprezentacji żądań HTTP.

6.6.1 Wnioski

Przeprowadzone badania eksperymentalne wykazały, że przy wykorzystaniu reprezentacji żądań HTTP narzędzia *Hfinger* algorytmy osiągały wartości miary F1 w zakresie od 0,24 do 0,94, natomiast miary MCC w zakresie od 0,18 do 0,97. Dla wszystkich algorytmów wraz ze wzrostem poziomu otwartości maleje wydajność klasyfikacji, choć dla algorytmu RF spadek jest znaczący w większości badanych przypadków i osiąga w nich wartości minimalne spośród wszystkich algorytmów np. przy wyborze hiperparametrów względem miary F1 wartość miary F1 spadła nawet o 69%, a miary MCC o 79%. Pod względem wydajności klasyfikacji dla różnych poziomów otwartości najlepszymi algorytmami są OSSVM oraz MCR-SVM, choć ten drugi od $O_P = 50\%$ osiąga większe spadki wydajności niż OSSVM. Mimo gorszych wyników niż OSSVM i MCR-SVM, algorytm EVM osiąga dobre wyniki klasyfikacji, które dla niektórych przypadków otwartości powyżej poziomu 25% są nawet 3-krotnie wyższe niż dla MCR-SVM.

Analiza czasów wykonania wyboru hiperparametrów, treningu i ewaluacji wykazała, że dla algorytmów W-SVM oraz OSSVM są one najdłuższe. Sumarycznie te trzy etapy zajmują im nawet 49 razy dłużej (w przypadku OSSVM dla $O_P = 0\%$ względem EVM). Algorytm W-SVM, mimo dobrych wyników miary F1 w scenariuszu wyboru hiperparametrów względem miary F1, które były porównywalne lub lepsze niż dla EVM, w przypadku miary MCC wypadł gorzej niż OSSVM, MCR-SVM lub EVM. Podobnie ma się sytuacja dla obu miar w scenariuszu wyboru hiperparametrów względem miary MCC. Dodając do tego wysokie czasy wykonania obliczeń, algorytm W-SVM można uznać za mało przydatny w porównaniu do np. EVM.

Analiza domyślnej reprezentacji żądań HTTP narzędzia *Hfinger* wykazała, że jej użycie umożliwia osiągnięcie dobrych wyników klasyfikacji, które nie odbiegają w dużym stopniu od wyników pełnej reprezentacji narzędzia *Hfinger*. Poziomy klasyfikacji w większości przypadków są takie same lub niższe niż w pełnej reprezentacji, podobnie jak sumaryczne czasy wykonania analizy. Największe różnice na niekorzyść domyślnej reprezentacji były widoczne dla algorytmu W-SVM, choć np. dla MCR-SVM dla $O_P = 0\%$ i $O_P = 50\%$ (wybór hiperparametrów względem F1) pełna reprezentacja była gorsza. Całościowo można stwierdzić, że zależności między algorytmami widoczne w analizie pełnej reprezentacji *Hfinger* zostały zachowane. Dobry poziom wydajności klasyfikacji badanych algorytmów wykorzystujących domyślną reprezentację *Hfinger*, nieodbiegający w dużym stopniu od wyników dla pełnej reprezentacji, pozwala stwierdzić, że wybór cech tworzących domyślną reprezentację narzędzia *Hfinger* został wykonany poprawnie. Niemniej, to pełna reprezentacja w większości przypadków umożliwia osiągnięcie lepszych wyników klasyfikacji przy stosunkowo małym wzroście czasu analizy.

Porównanie pełnej reprezentacji żądań HTTP narzędzia *Hfinger* oraz reprezentacji trigramowej wykazało, że dla otwartości do 25% to pełna reprezentacja *Hfinger* osiąga dużo lepsze wyniki. Przykładowo dla algorytmu EVM miara F1 jest większa od 18% do 48%, a dla MCR-SVM od 46% do 74%. Jednak od otwartości na poziomie 50% dochodzi do zmiany i reprezentacja trigramowa osiąga nawet 21% lepsze wyniki miary F1 dla algorytmu EVM i 80% lepsze dla MCR-SVM. Niestety, użycie reprezentacji trigramowej powoduje znaczące zwiększenie zapotrzebowania na pamięć operacyjną oraz zasoby obliczeniowe. Spowodowało to znaczące zwiększenie czasów wykonania analizy. Dla algorytmu EVM czas wyboru hiperparametrów wzrósł nawet 38 razy, etap treningu zajmował nawet o 214% więcej czasu, a czas ewaluacji był dłuższy maksymalnie 145 razy. Algorytm MCR-SVM dokonywał wyboru hiperparametrów nawet 29 razy dłużej, trening zajmował maksymalnie 22 razy więcej czasu, a ewaluacja nawet

17 razy dłużej.

Podsumowując, przedstawione wyniki przeprowadzonych badań wykazały, że otwartozbiorowe algorytmy uczenia maszynowego pozwalają osiągnąć lepsze wyniki klasyfikacji niż tradycyjne algorytmy, co widać na przykładzie algorytmu OSSVM. Niestety, nie wszystkie algorytmy radzą sobie równie dobrze, czego przykładem jest algorytm RF. Algorytm MCR-SVM, mimo że nie został skonstruowany jako algorytm otwartozbiorowy, dla niskich poziomów otwartości osiąga dobre wyniki klasyfikacji w stosunku do wyników algorytmów tego rodzaju. Wzięcie pod uwagę czasu wykonania analizy powoduje, że algorytm EVM, który osiąga gorsze wyniki klasyfikacji niż OSSVM lub częściowo MCR-SVM, staje się rozsądnym wyborem ze względu na stosunkowo szybkie działanie.

Wykorzystanie reprezentacji żądań narzędzia *Hfinger* umożliwia osiągnięcie dobrych wyników klasyfikacji oraz krótkich czasów analizy danych w porównaniu do reprezentacji trigramowej. Niemniej jednak przy wzroście otwartości powyżej 50% reprezentacja trigramowa umożliwia osiągnięcie lepszej wydajności. Niestety jej dużym minusem są duże wymagania dotyczące pamięci RAM oraz zasobów obliczeniowych, co w praktyce przekłada się na znaczne wydłużenie czasu analiz, w pewnych przypadkach nawet o dwa rzędy wielkości.

Ostateczną rekomendacją odnośnie tworzenia systemu klasyfikacji złośliwego oprogramowania bazującego na żądaniach HTTP jest wykorzystanie algorytmów EVM oraz MCR-SVM bazujących na pełnej reprezentacji *Hfinger*. Umożliwi to stosunkowo szybkie zapoznanie się z wynikami klasyfikacji oraz dostosowanie systemu do zbioru danych, który jest analizowany. Niemniej należy wziąć pod uwagę, że spośród tych dwóch, tylko algorytm EVM jest algorytmem otwartozbiorowym, zapewniającym ograniczenie ryzyka otwartozbiorowego. Rekomendowane algorytmy można wykorzystać do zbudowania klasyfikatora zespołowego, łączącego wyniki z obydwu, ponieważ tak jak MCR-SVM osiąga lepsze wyniki dla małych wartości otwartości, tak EVM jest lepszy dla większych. Niemniej badania takie wykraczają już poza ramy niniejszej rozprawy.

6.6.2 Dyskusja

Pewne elementy przeprowadzonych badań mogą zostać poprawione. Po pierwsze, jak w każdym badaniu wykorzystującym algorytmy uczenia maszynowego, ważnym elementem jest jakość wykorzystywanego zbioru danych. Ten użyty w prezentowanej analizie został stworzony przez autora na bazie m.in. rzeczywistych źródeł ruchu sieciowego złośliwego oprogramowania oraz

poddany procesowi oznaczania przy użyciu specjalistycznego zestawu reguł IDS ET Pro. Niestety, tak stworzony zbiór danych zawiera tylko te rodziny złośliwego oprogramowania, które są rozpoznawane przez użyty zestaw reguł. Dodatkowo zbiór danych odzwierciedla charakter źródeł, z których pochodzi, czyli przede wszystkim środowiska analizy złośliwego oprogramowania zespołu CERT Polska oraz projektu Stratosphere IPS uczelni CTU (Czech Technical University). Z tego powodu wyniki otrzymane dla użytego zbioru danych mogą się różnić przy wykorzystaniu zbioru danych pochodzących z innego środowiska.

Po drugie liczba przeanalizowanych algorytmów uczenia maszynowego, zarówno otwartozbiorowych jak i tradycyjnych, została ograniczona ze względu na możliwości techniczne oraz czasowe badań. W każdej analizie konieczne jest ograniczenie zakresu badanych elementów. Nie inaczej było w tej prezentowanej powyżej. Jako że według wiedzy autora jest to pierwsza próba wykorzystania algorytmów otwartozbiorowych do wykrywania złośliwego oprogramowania przy użyciu żądań HTTP, musiały zostać podjęte decyzje o ograniczeniu obiektów badań, aby otrzymać początkowe wyniki w tej tematyce badań. Niemniej nic nie stoi na przeszkodzie, żeby na bazie obecnych wyników rozszerzyć analizę o kolejne algorytmy.

Po trzecie, wykorzystywane w badaniach miary klasyfikacji F1 i MCC nie przypisują dużej wagi do informacji o wydajności klasyfikacji dla klas nieznanych. W początkowych etapach badań wykorzystywano dodatkową miarę HNA (Harmonic Normalized Accuracy) zaproponowaną przez Júniora i in. w [54]. Miara HNA jest średnią harmoniczną dokładności (accuracy) dla klas znanych oraz dokładności dla klas nieznanych. Niestety, ze względu na wieloklasowy i niezrównoważony pod względem liczby próbek w klasach charakter zbioru danych, miara ta w praktyce nie odzwierciedlała w poprawny sposób wydajności klasyfikacji. Głównym powodem jest wykorzystanie miary dokładności, która nie daje dobrych wyników dla zbiorów z dużymi różnicami w liczbie próbek (por. [43] s. 109). Z tego powodu zrezygnowano z użycia miary HNA.

Kolejną kwestią jest ograniczona liczba poziomów otwartości, dla których wykonano badania. Ze względu na dużą liczbę czynników analiz, jak np. liczba algorytmów, różne reprezentacje żądań czy wybór hiperparametrów, zdecydowano się na przebadanie jedynie pięciu wartości otwartości. Liczbę tę można zwiększyć, umieszczając punkty analizy między już wyznaczonymi, ale także wykraczając ponad poziom 75% wykorzystany w badaniach.

Pewne problemy można dostrzec w protokole badań. Jego projektowi przyświecał pomysł minimalizacji obciążenia estymacji efektywności klasyfikatora, przy równoczesnym ograni-

czeniu liczby koniecznych klasyfikacji. Wybrany schemat procedury zagnieżdżonego wyboru hiperparametrów lub powtórzenia podziału zbioru danych mogą zostać wykonane większą liczbę razy, lub zastąpione całkowicie innymi schematami. Dosyć ważnym elementem wyboru hiperparametrów jest zastosowanie podziału zestawu danych na 5 równych grup klas nieznanych i wykonania analiz dla każdej grupy. Ten schemat podziału może zostać także zmieniony pod względem liczności grup, a także całkowitego wyeliminowania na rzecz innego podejścia.

Pozostając przy tematyce hiperparametrów, do ich wyboru wykorzystana została metoda *HalvingGridSearchCV*, dzięki której udało się skrócić czas wykonania poszukiwań w stosunku do tradycyjnej metody *GridSearchCV*. Niemniej inne metody poszukiwania optymalnych hiperparametrów (dostępne np. w pakiecie Optuna[40]) mogą okazać wydajniejsze i dać lepsze efekty.

Wreszcie dosyć ważną kwestią, która nie była rozpatrywana w niniejszych badaniach, jest poziom otwartości analizowanych danych. W syntetycznych warunkach laboratoryjnych poziom ten był ustalany odgórnie i całkowicie kontrolowany. W rzeczywistych warunkach otwartość badanego zbioru danych jest nieznana, niemniej ustalenie jej wartości może być konieczne ze względu na charakter swoistego hiperparametru wykorzystywanego przez algorytmy MCR-SVM, EVM, W-SVM i RF. Poznanie jej wartości może być nietrywialnym zadaniem i konieczne może być szacowanie jej wartości.

6.6.3 Dalsze prace

Perspektywiczne kierunki dalszej pracy obejmują przebadanie innych reprezentacji żądań HTTP, w tym innych trybów działania narzędzia *Hfinger*. Podobnie z reprezentacją n-gramową: warte dalszej analizy są inne wartości niż trigramy, np. 5-gramy, ale przy zmniejszeniu liczby cech.

Interesującym kierunkiem badań jest wykorzystanie jeszcze innych algorytmów otwartozbiorowych, a także standardowych, bazujących na uczeniu zespołowym, jak np. XGBoost[41]. C. Geng i in. [42] dokonali przeglądu algorytmów otwartozbiorowych, w którym wskazali, że dostępne są inne podejścia niż przebadane w niniejszej pracy, które także mogą wykorzystać reprezentacje żądań bazujące na narzędziu *Hfinger* lub n-gramach. Można także spróbować tworzyć zespoły klasyfikatorów z tych już zbadanych i dokonać analizy ich wydajności klasyfikacji.

Z rozszerzeniem listy badanych algorytmów łączy się perspektywa poprawienia wydajności

[40] `https://optuna.org`

[41] `https://github.com/dmlc/xgboost`

obliczeniowej algorytmów przebadanych w powyższej analizie, w szczególności OSSVM. W tej sytuacji możliwe stałoby się porównanie tego algorytmu na tle np. zespołu klasyfikatorów SVM w scenariuszu jeden przeciwko wszystkim.

Więcej analiz może się przysłużyć protokołowi badań oraz miarom klasyfikacji. Zaprezentowany protokół badań nie jest pozbawiony wad, jak np. wykorzystanie arbitralnego poziomu otwartości przy wyborze hiperparametrów, dlatego można dokonać jego ulepszeń. Podobnie z miarami klasyfikacji. Użyte w badaniach F1 i MCC nie oddają w pełni wydajności klasyfikacji klas nieznanych i jak zostało to omówione powyżej, miary stworzone specjalnie dla scenariusza otwartozbiorowego, jak np. HNA, w praktyce nie sprawdzają się w zbiorach danych niezrównoważonych pod względem liczby próbek w klasach.

Wreszcie, w związku z szybkim rozwojem tematyki sieci neuronowych, warta sprawdzenia jest ich przydatność do zagadnienia poruszanego w niniejszej rozprawie. Łączyłoby się to ze zmianą podejścia do reprezentacji żądań, np. ich zamiany na mapy bitowe i przetwarzania jako obrazów.

7 Podsumowanie

W poniższym rozdziale znajduje się podsumowanie prac badawczych zaprezentowanych w rozprawie oraz ich krytyczna analiza. Zarysowano również potencjalne kierunki dalszych badań.

7.1 Podsumowanie wyników prac

W celu udowodnienia tezy rozprawy sformułowanej w rozdziale 1.3. została ona podzielona na trzy części składowe – podtezy, których udowodnieniem zajmowały się rozdziały 4 - 6. Każda z nich została podsumowana poniżej.

1. **Ruch sieciowy żądań protokołu HTTP złośliwego oprogramowania zawiera cechy charakterystyczne, które umożliwiają jego odróżnienie od ruchu sieciowego aplikacji niezłośliwych.**

   W rozdziale 4. przedstawiono badania żądań HTTP złośliwego oprogramowania oraz aplikacji niezłośliwych reprezentowanych przez popularne przeglądarki internetowe. Celem tych badań była identyfikacja cech charakterystycznych, umożliwiających odróżnienie tych dwóch typów aplikacji. Analizowane cechy zostały sformułowane na bazie prac innych badaczy oraz własnego doświadczenia autora, zdobytego przy analizie złośliwego oprogramowania. Uzyskane rezultaty wykazały, że część z badanych cech występuje częściej w ruchu sieciowym złośliwego oprogramowania. Są to: użycie wersji 1.0 protokołu, liczba nagłówków mniejsza niż 4, brak w żądaniu nagłówków *Accept*, *Accept-Encoding*, *Accept-Language*, *Connection* oraz do pewnego stopnia nagłówka *Referer*, entropia przesyłanych danych większa niż 6 bitów, obecność znaków nie-ASCII w przesyłanych danych oraz obecność nagłówka *Referer* w żądaniu typu *POST*. Badania wartości nagłówka *Host* wykazały, że wartości inne niż nazwa domeny, jak np. adres IP, są częściej spotykane w ruchu sieciowym złośliwego oprogramowania. Niemniej jednak wykorzystanie tego nagłówka do detekcji powinno być przeprowadzane z odpowiednią dbałością, ponieważ jego wartość jest ściśle zależna od infrastruktury używanej przez przestępców. Przykładowo, zmiana wykorzystywanego adresu IP na domenę, może zakłócić proces detekcji. Z kolei wykorzystanie wartości nagłówka *User-Agent* do identyfikacji złośliwego oprogramowania jest dosyć złożoną kwestią. Wiele rodzin złośliwego oprogramowania podszywa się pod popularne aplikacje niezłośliwe używając tych samych lub podobnych łańcuchów znakowych. Choć istnieje też grupa rodzin, które wykorzystują wartości odmienne od

tych popularnych. Wreszcie brak tego nagłówka w żądaniu powinien zostać uznany za anomalię i mocny wskaźnik złośliwości żądania. Inne badane cechy były zbyt rzadkie lub w ogóle nie występowały w badanym ruchu sieciowym. Niemniej jednak mogą być traktowane jako anomalie same w sobie ze względu na niespójność z RFC. Wśród nich można wymienić: brak dwukropka w polu nagłówka, literówki w nazwie nagłówka, wysyłanie żądań na port inny niż zarejestrowany dla HTTP, obecność znaków nie-ASCII w wartości nagłówka, znak nowej linii inny niż CRLF, powtórzenie nagłówka czy niestandardowy biały znak inny niż spacja lub tabulacja horyzontalna.

Jak można wywnioskować z powyższego podsumowania, ruch sieciowy protokołu HTTP złośliwego oprogramowania zawiera pewne cechy charakterystyczne, które umożliwiają jego odróżnienie od ruchu sieciowego aplikacji niezłośliwych. Oznacza to, że analizowana podteza została udowodniona.

2. **Można stworzyć unikalną reprezentację żądań protokołu HTTP, która umożliwia identyfikację złośliwego oprogramowania.**

   W rozdziale 5. znajduje się opis badań narzędzia *Hfinger*, umożliwiającego stworzenie reprezentacji żądań protokołu HTTP w formie zrozumiałej przez człowieka. Reprezentacje te można nazwać *odciskami* (*fingerprint*). *Hfinger* wykorzystuje informacje występujące w różnych częściach żądania: adres URL, informacje sterujące protokołu, nagłówki i ich wartości oraz przesyłane dane. Na tej bazie tworzona jest unikalna reprezentacja. *Hfinger* został porównany z innymi popularnymi narzędziami o podobnych możliwościach: *FATT*, *Mercury* oraz *p0f*. W badaniach analizowano ruch sieciowy złośliwego oprogramowania oraz popularnych aplikacji niezłośliwych. Porównywano unikalność tworzonych odcisków definiowaną na bazie liczby ich powtórzeń dla różnych rodzin złośliwego oprogramowania, a także dla aplikacji niezłośliwych. Sprawdzano także wydajność generalizacji informacji z żądań przez zliczanie liczby tworzonych odcisków, a także ich entropię. *Hfinger* wykorzystuje jeden z 5 trybów pracy, które są zdefiniowane przez używany podzbiór cech żądań HTTP. Wybór tych podzbiorów został dokonany w ramach przeprowadzonych prac badawczych w celu optymalizacji efektywności tworzenia reprezentacji. Domyślny tryb działania narzędzia umożliwia tworzenie odcisków będących kompromisem między ich unikalnością a ich liczbą. Inne tryby pracy osiągają lepszą entropię odcisków, zmniejszają ich liczbę lub wykorzystują większą liczbę cech żądania. Badania wykazały, że *Hfinger* w domyślnym trybie pracy generuje od 8 do 34 razy bardziej unikalne odciski niż inne

popularne narzędzia tego typu. W przeciwieństwie do nich, w trybie tym nie wprowadza żadnych powtórzeń odcisków między złośliwym oprogramowaniem i popularnymi aplikacjami niezłośliwymi. Porównując do narzędzi *FATT* i *Mercury*, *Hfinger* generuje co najwyżej trzy razy więcej odcisków, ale nadal 35% mniej niż *p0f*. Uzyskane przez *Hfinger* poziomy entropii stworzonych reprezentacji są wyższe niż dla *FATT* i *Mercury* i tylko trochę niższe niż dla *p0f*.

Hfinger można stosować w różnych scenariuszach. Dzięki wykorzystaniu plików pcap ze zrzutem ruchu sieciowego narzędzie może być używane w systemach typu honeypot, sandboksach, ale też w sieciach domowych lub korporacyjnych. Tworzone odciski mogą być użyte do bezpośredniej analizy śledczej ruchu sieciowego oraz jako źródło danych do wyspecjalizowanych systemów logowania i monitorowania zdarzeń sieciowych (np. klasy SIEM - Security Information and Event Management). *Hfinger* można wykorzystać do grupowania podobnych żądań, dzięki czemu staje się możliwe śledzenie działań złośliwego oprogramowania. Tworzenie unikalnych reprezentacji umożliwia także identyfikację operacji malware'u, jak np. sprawdzenia adresu IP lub próby połączenia do serwera C&C. *Hfinger* sam nie umożliwia detekcji ani klasyfikacji złośliwego oprogramowania, niemniej dzięki swojej unikalności, tworzone odciski można oznaczyć np. nazwą rodziny złośliwego oprogramowania i wykorzystać do identyfikacji przeszłego lub przyszłego ruchu sieciowego. Narzędzie *Hfinger* jest publicznie dostępne w serwisie GitHub[42]. Używane jest również w środowisku automatycznej analizy malware'u zespołu CERT Polska (realizującego zadania CSIRT NASK jako jednego z trzech zespołów CSIRT poziomu krajowego według ustawy o krajowym systemie cyberbezpieczeństwa).

Narzędzie *Hfinger* umożliwia tworzenie unikalnej reprezentacji żądań protokołu HTTP w efektywniejszy sposób niż inne popularne narzędzia. Jak omówiono to powyżej, tworzone oznaczenie żądań umożliwia odróżnienie od siebie odmiennych rodzin złośliwego oprogramowania, a także jego różnych operacji. Dzięki temu możliwa staje się jego identyfikacja. Tym samym omawiana podteza rozprawy została udowodniona.

3. **Wykorzystanie odpowiednio stworzonej reprezentacji żądań HTTP umożliwia skuteczne rozpoznanie rodzin złośliwego oprogramowania, w tym także istnienia klas dotychczas nieznanych.**

[42] `https://github.com/CERT-Polska/hfinger`

Badania zamieszczone w rozdziale 6. porównują wydajność klasyfikacji otwartozbiorowej złośliwego oprogramowania przy użyciu żądań protokołu HTTP na bazie różnych algorytmów uczenia maszynowego. Uwzględnione zostały trzy algorytmy otwartozbiorowe: W-SVM, OSSVM i EVM oraz dwa klasyczne algorytmy wyposażone w mechanizm odrzucania predykcji o zbyt małym prawdopodobieństwie: SVM (oznaczany jako MCR-SVM) oraz Random Forest (oznaczany jako RF). Wykorzystano również dwie reprezentacje żądań bazujące na narzędziu *Hfinger* oraz reprezentację wykorzystująca analizę n-gramową. Przeprowadzone badania wykazały, że otwartozbiorowe algorytmy uczenia maszynowego osiągają lepsze poziomy skuteczności klasyfikacji niż tradycyjne metody. Przykładem jest rozwiązanie OSSVM, które przy użyciu pełnej reprezentacji żądań HTTP narzędzia *Hfinger* uzyskiwało bardzo dobre wartości miar F1 i MCC, lepsze lub zbliżone do wyników najlepszego tradycyjnego algorytmu MCR-SVM. Niestety jego długie czasy analizy powodują, że konieczna jest jego optymalizacja w celu zwiększenia wydajności obliczeniowej. Mimo że algorytm MCR-SVM nie jest algorytmem otwartozbiorowym, to dla niskich poziomów otwartości zbioru danych (czyli liczby klas nieznanych) osiąga dobre poziomy klasyfikacji, lepsze niż algorytmy otwartozbiorowe inne niż OSSVM. Jednak ze wzrostem poziomu otwartości różnica miedzy nim a algorytmem otwartozbiorowym EVM zanika. To właśnie EVM jest rozsądnym wyborem biorąc pod uwagę skuteczność klasyfikacji oraz czas przeprowadzenia analiz.

Porównanie wykorzystanych reprezentacji narzędzia *Hfinger* wykazało, że różnica między nimi jest minimalna. Zależności pomiędzy poziomami skuteczności badanych algorytmów są podobne dla obu reprezentacji. Niemniej jednak, w większości przypadków to pełna reprezentacja narzędzia *Hfinger* jest lepsza pod względem klasyfikacji w porównaniu do domyślnej reprezentacji. Dzieje się to przy stosunkowo małym wzroście czasu analizy.

Z kolei porównanie reprezentacji żądań bazującej na trigramach do pełnej reprezentacji narzędzia *Hfinger* wykazało, że ta ostatnia osiąga lepsze wyniki klasyfikacji dla otwartości do 25%. Przykładowo, dla algorytmu EVM miara F1 była wyższa od 18% do 48%, a dla MCR-SVM od 46% do 74%. Powyżej tej wartości sytuacja się odwraca i reprezentacja trigramowa okazuje się lepsza: osiąga nawet o 21% wyższe poziomy miary F1 dla algorytmu EVM oraz o 80% wyższe dla algorytmu MCR-SVM. Niestety wykorzystanie analizy n-gramowej powoduje zwiększenie zapotrzebowania na pamięć RAM oraz moc obliczeniową ze względu na konieczność przetwarzania danych o większej wymiarowości niż w

przypadku reprezentacji narzędzia *Hfinger*. Jest to odzwierciedlone w znaczących wzrostach czasu analizy zaobserwowanych przy wykorzystaniu trigramów. Przykładowo, dla algorytmu EVM czas wyboru hiperparametrów wzrósł nawet 38-krotnie, a czas ewaluacji nawet 145-krotnie. Na podstawie przeprowadzonych badań i uzyskanych rezultatów, przy wyborze algorytmu uczenia maszynowego dla systemu klasyfikacji umożliwiającego rozpoznawanie otwartozbiorowe należy wziąć pod uwagę EVM oraz MCR-SVM. Niestety, ten drugi jest algorytmem zamkniętozbiorowym, co powoduje, że w jego przypadku ryzyko otwartozbiorowe jest nieograniczone. Jest to widoczne pod postacią mocnego spadku skuteczności klasyfikacji przy zwiększeniu otwartości zbioru, czyli wzroście liczby nieznanych rodzin złośliwego oprogramowania. Rozwiązaniem w tym przypadku mogłoby być stworzenie klasyfikatora zespołowego, łączącego wyniki tych dwóch algorytmów.

Ostatecznie, przeprowadzone badania wykazały, że reprezentacja żądań HTTP tworzona przez narzędzie *Hfinger* może być wykorzystana do skutecznej klasyfikacji otwartozbiorowej złośliwego oprogramowania. Osiągnięte dzięki tej reprezentacji poziomy klasyfikacji są lepsze niż dla alternatywnej reprezentacji żądań bazującej na analizie trigramowej dla otwartości zbioru danych mniejszych lub równych 25%. Powyżej tej wartości reprezentacja trigramowa osiąga lepsze poziomy klasyfikacji. Niestety czasy analizy tej reprezentacji są nawet kilkudziesięciokrotnie większe. Z tej perspektywy należy uznać, że omawiana podteza została udowodniona.

Jak zaprezentowano powyżej, wszystkie części składowe głównej tezy zostały udowodnione. Z tego powodu należy uznać, że główna teza rozprawy także została udowodniona: analiza żądań protokołu HTTP umożliwia identyfikację złośliwego oprogramowania, a także klasyfikację jego rodzin w scenariuszu otwartozbiorowym.

7.2 Dyskusja

Autor niniejszej rozprawy dołożył wszelkich starań by przeprowadzone badania i analizy były poprawne. Niemniej jednak można podnieść wobec nich kilka zastrzeżeń.

Wnioski wyciągnięte z przeprowadzonych badań są ściśle zależne od wykorzystanych zbiorów danych. Zbiory te zostały specjalnie stworzone na potrzeby niniejszej rozprawy. Do zestawu danych zawierających ruch sieciowy aplikacji niezłośliwych wykorzystano m.in. popularne oprogramowanie biurowe czy popularne przeglądarki internetowe. W sytuacji wykorzystania

w monitorowanej sieci mniej popularnych aplikacji, które generowałyby inny ruch sieciowy, część wyciągniętych wniosków mogłaby także ulec zmianie. Z kolei w części zbioru danych dotyczącej złośliwego oprogramowania, wykorzystany ruch sieciowy został oznaczony głównie przy użyciu reguł IDS ET Pro. Są one poważanym i często wykorzystywanym w środowisku bezpieczeństwa źródłem detekcji zagrożeń. Jednak możliwe jest, że w trakcie tego procesu mogło dojść do sytuacji, gdy nie wszystkie złośliwe żądania zostały odkryte ze względu na brak odpowiedniej reguły detekcji, a także niektóre żądania zostały oznaczone niepoprawną nazwą. Dodatkowo, mimo użycia szerokiej reprezentacji różnych rodzin złośliwego oprogramowania, zbiór danych nie zawierał wszystkich możliwych rodzin. Nie jest wykluczone, że istnieją rodziny, które prezentują inne cechy ruchu sieciowego niż zidentyfikowane w powyższych badaniach, co mogłoby prowadzić do wyciągnięcia innych wniosków niż zaprezentowane powyżej. Niemniej jednak, w związku z przebadaniem w niniejszej rozprawie wielu popularnych rodzin malware oraz dosyć spójnym występowaniem w nich cech, można z dużą dozą prawdopodobieństwa stwierdzić, że zaprezentowane analizy można w dużym stopniu uogólnić także na inne rodziny złośliwego oprogramowania.

W zaprezentowanych badaniach skupiono się na środowisku systemu operacyjnego Windows. Poczynione obserwacje można w pewnym stopniu przenieść na inne platformy, np. mobilne. Z pewnością poprawne pozostaje wykorzystane w badaniach podejście do analiz, które można w łatwy sposób zastosować do zbiorów danych innych systemów operacyjnych.

Twórcy złośliwego oprogramowania mogą próbować obejść mechanizmy identyfikacji narzędzia *Hfinger* lub systemów detekcji bazujących na algorytmach uczenia maszynowego. W pierwszym przypadku, cyberprzestępcy mogą starać się wykorzystać mechanizmy mimikry i upodobnić żądania do aplikacji niezłośliwych lub innych rodzin złośliwego oprogramowania. Jest to z pewnością ciekawy kierunek przyszłych badań. Niemniej jednak w opinii autora niniejszej rozprawy pełna mimikra może być trudna lub niemożliwa do zrealizowania. Spowodowane jest to głównie odmiennym charakterem działania, a przez to sposobem generowania ruchu sieciowego między różnymi typami aplikacji. W związku z tym, należy się spodziewać, że nadal będą widoczne pewne elementy charakterystyczne odróżniające złośliwe oprogramowanie od aplikacji niezłośliwych. Problem obejścia mechanizmów narzędzia *Hfinger* ściśle oddziałuje na systemy detekcji bazujące na nim. Upodobnianie żądań może wpływać na poziom skuteczności klasyfikacji i możliwości odróżniania rodzin. Można to uznać za jeden ze sposobów ataku na systemy detekcji wykorzystujące algorytmy uczenia maszynowego. Sposoby ataków i obrony na

tego typu systemy są aktywnym polem badawczym (por. I. Rosenberg i in. [86]). Analiza możliwych ataków i sposobów obrony przed nimi w perspektywie klasyfikatora otwartozbiorowego wykorzystującego protokół HTTP jest zagadnieniem wartym dalszego zbadania. Potencjalnymi kierunkami w tym przypadku jest np. detekcja manipulacji danych wejściowych klasyfikatora, choćby przez mimikrę żądań.

W badaniach przeprowadzonych w ramach niniejszej rozprawy skupiono się na protokole HTTP w wersji 1.1. W sytuacji, gdy złośliwe oprogramowanie korzystałoby z wyższych wersji tego protokołu, przeprowadzone analizy można do nich zaadaptować, np. poprzez dekodowanie przesyłanych danych i zamianę ich na postać tekstową. Z kolei w sytuacji wykorzystania protokołu HTTPS, wykorzystanie opracowanych systemów jest utrudnione ze względu na zaszyfrowaną postać danych. Niemniej jednak w wielu sieciach, jak np. środowiskach korporacyjnych lub zaawansowanych systemach sandboksowych, ruch protokołu HTTPS jest deszyfrowany do postaci jawnej, dzięki czemu analiza bazująca na protokole HTTP jest cały czas możliwa.

W końcu do przeprowadzonych analiz można wykorzystać lepsze miary badawcze, lepiej odzwierciedlające badane zjawiska, jak np. poprawne wykrycie klas nieznanych. Pozostając przy klasyfikacji otwartozbiorowej, można także ulepszyć protokół badań w celu lepszej generalizacji estymacji wyników klasyfikacji.

7.3 Dalsze prace badawcze

Wykonane prace badawcze mają kilka potencjalnych, ciekawych kierunków do dalszej kontynuacji.

Jak już zostało wspomniane, przeprowadzone badania wykonane były dla środowiska systemu operacyjnego Windows. Naturalnym rozszerzeniem jest zaadoptowanie do platform mobilnych, jak np. system Android, a także innych systemów bazujących na środowisku Linuksa, które obejmuje również część systemów IoT (Internet of Things).

Warto także rozszerzyć możliwości narzędzia *Hfinger*. Z jednej strony poprzez wprowadzenie mechanizmów poszukiwania przybliżonego (fuzzy-searching), dzięki czemu mogłyby wzrosnąć możliwości generalizujące oraz identyfikacyjne narzędzia. Innym interesującym kierunkiem badawczym byłoby rozszerzenie zakresu danych poddawanych analizie, w tym np. zwiększenie liczby cech bazujących na danych przesyłanych w żądaniu (payload).

Ważnym kierunkiem prac jest wykorzystanie do klasyfikacji sieci neuronowych. W takim przypadku możliwa byłaby rezygnacja z konieczności tworzenia i utrzymywania cech klasyfi-

katora na rzecz bezpośredniego wykorzystania surowych danych żądań, np. w postaci obrazów. W podobny sposób przetwarzane są dane przepływów w Z. Liu i in. [70] lub Y. Zhang i in. [116]. Natomiast zastosowanie do tego warstwy OpenMax umożliwiałoby pracę w scenariuszu otwartozbiorowym.

W prezentowanych badaniach zrezygnowano z wykorzystania odpowiedzi HTTP ze względu na niepewność co do ich poprawności. Interesującym kierunkiem badań byłoby użycie tych odpowiedzi pomimo tej niepewności. System taki mógłby tworzyć modele par żądanie - odpowiedź lub zbiorów takich par i na ich bazie dokonywać klasyfikacji. Przy czym mógłby wskazywać na niepoprawność odpowiedzi razem z powodem tej niepoprawności, np. inny kod odpowiedzi, odmiennie zwrócone dane lub inne wartości nagłówków.

Z kolei scenariusz otwartozbiorowy umożliwia wykrywanie rodzin złośliwego oprogramowania nieznanych w czasie treningu klasyfikatora. Jednak gdy informacja ta już się pojawi, warto by została wykorzystana przez system do poprawienia klasyfikacji. Może w tym pomóc podejście uczenia przyrostowego, które jest wspierane np. przez jeden z wykorzystanych w tej rozprawie algorytmów – EVM. Kolejne kroki badawcze mogą być skierowane w kierunku optymalnego wykorzystania tego typu podejścia.

Wreszcie ważnym kierunkiem jest analiza potencjalnych ataków na klasyfikator bazujący na algorytmie uczenia maszynowego. Jak wspomniano w dyskusji w poprzednim podrozdziale, ataki te mogą powodować zmniejszenie wydajności klasyfikacji. Problem jest dosyć złożony, ponieważ zależy od użytej reprezentacji żądań oraz wykorzystanych algorytmów uczenia maszynowego. Dodatkowym utrudnieniem może być praca w scenariuszu otwartozbiorowym, co otwiera dodatkowe pole do badań.

Literatura

[1] D. Acarali, M. Rajarajan, N. Komninos, and I. Herwono, "Survey of approaches and features for the identification of HTTP-based botnet traffic," *Journal of Network and Computer Applications*, vol. 76, pp. 1–15, Dec. 2016. DOI: 10.1016/j.jnca.2016.10.007. [Online]. Available: https://doi.org/10.1016/j.jnca.2016.10.007.

[2] N. Agarwal and S. Z. Hussain, "A closer look at intrusion detection system for web applications," *Security and Communication Networks*, vol. 2018, pp. 1–27, Aug. 2018. DOI: 10.1155/2018/9601357. [Online]. Available: https://doi.org/10.1155/2018/9601357.

[3] H. Bai, G. Liu, W. Liu, Y. Quan, and S. Huang, "N-gram, semantic-based neural network for mobile malware network traffic detection," *Security and Communication Networks*, vol. 2021, Q. Liu, Ed., pp. 1–17, Apr. 23, 2021, ISSN: 1939-0122, 1939-0114. DOI: 10.1155/2021/5599556. [Online]. Available: https://www.hindawi.com/journals/scn/2021/5599556/.

[4] K. Baker, *The 11 Most Common Types of Malware*, Dostęp: 19.12.2021. [Online]. Available: https://www.crowdstrike.com/cybersecurity-101/malware/types-of-malware/.

[5] K. Bartos, M. Sofka, and V. Franc, "Optimized invariant representation of network traffic for detecting unseen malware variants," in *Proceedings of the 25th USENIX Conference on Security Symposium*, ser. SEC'16, Austin, TX, USA: USENIX Association, 2016, pp. 807–822.

[6] P. Bazydło, K. Lasota, and A. Kozakiewicz, "Botnet fingerprinting: Anomaly detection in smtp conversations," *IEEE Security & Privacy*, vol. 15, no. 6, pp. 25–32, 2017. DOI: 10.1109/MSP.2017.4251116.

[7] D. Bekerman, B. Shapira, L. Rokach, and A. Bar, "Unknown malware detection using network traffic classification," in *2015 IEEE Conference on Communications and Network Security (CNS)*, Florence, Italy: IEEE, Sep. 2015, pp. 134–142. DOI: 10.1109/CNS.2015.7346821. [Online]. Available: http://ieeexplore.ieee.org/document/7346821/.

[8] M. Belshe, R. Peon, and M. Thomson, *Hypertext Transfer Protocol Version 2 (HTTP/2)*, RFC 7540, May 2015. DOI: 10.17487/RFC7540. [Online]. Available: https://rfc-editor.org/rfc/rfc7540.txt.

[9] A. Bendale and T. E. Boult, "Towards open set deep networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[10] T. Berners-Lee, R. Fielding, and H. Frystyk, *RFC 1945: Hypertext Transfer Protocol – HTTP/1.0*, 1996.

[11] P. Białczak and W. Mazurczyk, "Characterizing anomalies in malware-generated HTTP traffic," *Security and Communication Networks*, vol. 2020, pp. 1–26, Sep. 2020. DOI: 10.1155/2020/8848863. [Online]. Available: https://doi.org/10.1155/2020/8848863.

[12] ——, "Hfinger: Malware HTTP request fingerprinting," *Entropy*, vol. 23, no. 5, 2021, ISSN: 1099-4300. DOI: 10.3390/e23050507. [Online]. Available: https://www.mdpi.com/1099-4300/23/5/507.

[13] M. Bishop, "Hypertext Transfer Protocol Version 3 (HTTP/3)," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-http-34, Feb. 2021, Work in Progress, 75 pp. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34.

[14] A. Blaise, M. Bouet, V. Conan, and S. Secci, "Botnet fingerprinting: A frequency distributions scheme for lightweight bot detection," *IEEE Transactions on Network and Service Management*, vol. 17, no. 3, pp. 1701–1714, 2020. DOI: 10.1109/TNSM.2020.2996502.

[15] R. Bortolameotti, T. van Ede, M. Caselli, *et al.*, "Decanter: Detection of anomalous outbound http traffic by passive application fingerprinting," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, ser. ACSAC 2017, Orlando, FL, USA: Association for Computing Machinery, 2017, pp. 373–386, ISBN: 9781450353458. DOI: 10.1145/3134600.3134605. [Online]. Available: https://doi.org/10.1145/3134600.3134605.

[16] R. Bortolameotti, T. van Ede, A. Continella, *et al.*, "Headprint: Detecting anomalous communications through header-based application fingerprinting," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, ser. SAC '20, Brno,

Czech Republic: Association for Computing Machinery, 2020, pp. 1696–1705, ISBN: 9781450368667. DOI: 10.1145/3341105.3373862. [Online]. Available: https://doi.org/10.1145/3341105.3373862.

[17] T. E. Boult, S. Cruz, A. Dhamija, M. Gunther, J. Henrydoss, and W. Scheirer, "Learning and the unknown: Surveying steps toward open world recognition," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 9801–9807, 2019. DOI: 10.1609/aaai.v33i01.33019801. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/5054.

[18] G. Bovenzi, G. Aceto, D. Ciuonzo, V. Persico, and A. Pescape, "A Hierarchical Hybrid Intrusion Detection Approach in IoT Scenarios," en, in *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, Taipei, Taiwan: IEEE, Dec. 2020, pp. 1–7, ISBN: 978-1-72818-298-8. DOI: 10.1109/GLOBECOM42002.2020.9348167. [Online]. Available: https://ieeexplore.ieee.org/document/9348167/.

[19] M. C. Calzarossa and L. Massari, "Analysis of header usage patterns of HTTP request messages," in *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS)*, IEEE, 2014, pp. 847–853.

[20] L. Caviglione, M. Choraś, I. Corona, *et al.*, "Tight arms race: Overview of current malware threats and trends in their detection," *IEEE Access*, vol. 9, pp. 5371–5396, 2021. DOI: 10.1109/ACCESS.2020.3048319.

[21] S. Chng, H. Y. Lu, A. Kumar, and D. Yau, "Hacker types, motivations and strategies: A comprehensive framework," *Computers in Human Behavior Reports*, vol. 5, p. 100 167, 2022, ISSN: 2451-9588. DOI: https://doi.org/10.1016/j.chbr.2022.100167. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S245195882200001X.

[22] S. Cruz, C. Coleman, E. M. Rudd, and T. E. Boult, "Open Set Intrusion Recognition for Fine-Grained Attack Categorization," en, *arXiv:1703.02244 [cs]*, Mar. 2017, arXiv: 1703.02244. [Online]. Available: http://arxiv.org/abs/1703.02244.

[23] *Cuckoo sandbox multiple User-Agents community module*, Dostęp: 29.03.2019 r. [Online]. Available: https://github.com/cuckoosandbox/community/blob/master/modules/signatures/windows/multiple_ua.py.

[24] *Cuckoo sandbox network CnC HTTP community module*, Dostęp: 29.03.2019 r. [Online]. Available: https://github.com/cuckoosandbox/community/blob/master/modules/signatures/network/network_cnc_http.py.

[25] Cybereason, *Ransomware: The true cost to business*, Dostęp: 27.03.2022 r. [Online]. Available: https://www.cybereason.com/hubfs/dam/collateral/ebooks/Cybereason_Ransomware_Research_2021.pdf.

[26] E. U. A. for Cybersecurity., *ENISA threat landscape 2021: April 2020 to mid July 2021*. Publications Office, 2021. DOI: 10.2824/324797. [Online]. Available: https://data.europa.eu/doi/10.2824/324797.

[27] A. Damodaran, F. D. Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 1, pp. 1–12, 2015. DOI: 10.1007/s11416-015-0261-z.

[28] A. Dolas, M. Sadique, and M. Ghule, *Catching RATs Over Custom Protocols*, Dostęp: 19.12.2021. [Online]. Available: https://www.zscaler.com/blogs/security-research/catching-rats-over-custom-protocols.

[29] B. Duncan, *RIG EK PUSHES GANDCRAB RANSOMWARE*, Dostęp: 19.12.2021. [Online]. Available: https://www.malware-traffic-analysis.net/2018/07/25/index.html.

[30] T. van Ede, R. Bortolameotti, A. Continella, *et al.*, "FlowPrint: Semi-supervised mobile-app fingerprinting on encrypted network traffic," in *Proceedings 2020 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2020, ISBN: 978-1-891562-61-7. DOI: 10.14722/ndss.2020.24412. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2020/02/24412.pdf.

[31] J. Ellis, *Evasion techniques: Geoblocking by ip*, Dostęp: 19.12.2021. [Online]. Available: https://www.phishlabs.com/blog/phishing-blocking-techniques-geoblocking-ip/.

[32] M. Eslahi, W. Z. Abidin, and M. V. Naseri, "Correlation-based HTTP botnet detection using network communication histogram analysis," in *2017 IEEE Conference on Application, Information and Network Security (AINS)*, Miri: IEEE, Nov. 2017, pp. 7–12. DOI:

10.1109/AINS.2017.8270416. [Online]. Available: http://ieeexplore.ieee.org/document/8270416/.

[33] M. Eslahi, M. S. Rohmad, H. Nilsaz, M. V. Naseri, N. Tahir, and H. Hashim, "Periodicity classification of HTTP traffic to detect HTTP botnets," in *2015 IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE)*, Langkawi, Kedah, Malaysia: IEEE, Apr. 2015, pp. 119–123. DOI: 10.1109/ISCAIE.2015.7298339. [Online]. Available: http://ieeexplore.ieee.org/document/7298339/.

[34] C. Fachkha, E. Bou-Harb, and M. Debbabi, "Fingerprinting internet dns amplification ddos activities," in *2014 6th International Conference on New Technologies, Mobility and Security (NTMS)*, 2014, pp. 1–5. DOI: 10.1109/NTMS.2014.6814019.

[35] R. Fielding, J. Gettys, J. Mogul, *et al.*, *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1*, 1999.

[36] R. Fielding, Y. Lafon, and J. Reschke, *RFC 7233: Hypertext Transfer Protocol (HTTP/1.1): Range requests*, 2014.

[37] R. Fielding, M. Nottingham, and J. Reschke, *RFC 7234: Hypertext Transfer Protocol (HTTP/1.1): Caching*, 2018.

[38] R. Fielding and J. Reschke, *RFC 7230: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*, 2014.

[39] ——, *RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*, 2014.

[40] ——, *RFC 7232: Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests*, 2014.

[41] ——, *RFC 7235: Hypertext Transfer Protocol (HTTP/1.1): Authentication*, 2014.

[42] C. Geng, S.-J. Huang, and S. Chen, "Recent advances in open set recognition: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 10, pp. 3614–3631, 2021. DOI: 10.1109/TPAMI.2020.2981604.

[43] A. Géron, *Uczenie maszynowe z użyciem Scikit-Learn i TensorFlow*. Helion, 2020, ISBN: 978-83-283-6003-7.

[44] M. Grill and M. Rehak, "Malware detection using HTTP user-agent discrepancy identification," in *2014 IEEE International Workshop on Information Forensics and Security (WIFS)*, Atlanta, GA, USA: IEEE, Dec. 2014, pp. 221–226, ISBN: 978-1-4799-8882-2. DOI: 10.1109/WIFS.2014.7084331. [Online]. Available: http://ieeexplore.ieee.org/document/7084331/.

[45] J. Guo, S. Guo, S. Ma, Y. Sun, and Y. Xu, "Conservative Novelty Synthesizing Network for Malware Recognition in an Open-Set Scenario," en, *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–15, 2021, ISSN: 2162-237X, 2162-2388. DOI: 10.1109/TNNLS.2021.3099122. [Online]. Available: https://ieeexplore.ieee.org/document/9503108/.

[46] F. Haddadi, J. Morgan, E. G. Filho, and A. N. Zincir-Heywood, "Botnet behaviour analysis using IP flows: With HTTP filters using classifiers," in *2014 28th International Conference on Advanced Information Networking and Applications Workshops*, BC, Canada: IEEE, May 2014, pp. 7–12. DOI: 10.1109/WAINA.2014.19. [Online]. Available: http://ieeexplore.ieee.org/document/6844605/.

[47] F. Haddadi and A. N. Zincir-Heywood, "A closer look at the HTTP and p2p based botnets from a detector's perspective," in *Foundations and Practice of Security*, J. Garcia-Alfaro, E. Kranakis, and G. Bonfante, Eds., vol. 9482, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2016, pp. 212–228. DOI: 10.1007/978-3-319-30303-1_13. [Online]. Available: http://link.springer.com/10.1007/978-3-319-30303-1_13.

[48] M. Hassen and P. K. Chan, "Learning a neural-network-based representation for open set recognition," in *Proceedings of the 2020 SIAM International Conference on Data Mining*, SIAM, 2020, pp. 154–162.

[49] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. Springer New York, 2009. DOI: 10.1007/978-0-387-84858-7. [Online]. Available: https://doi.org/10.1007/978-0-387-84858-7.

[50] J. Henrydoss, S. Cruz, E. M. Rudd, M. Gunther, and T. E. Boult, "Incremental Open Set Intrusion Recognition Using Extreme Value Machine," en, in *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, Cancun: IEEE, Dec. 2017, pp. 1089–1093, ISBN: 978-1-5386-1418-1. DOI: 10.1109/ICMLA.

2017.000-3. [Online]. Available: https://ieeexplore.ieee.org/document/8260789/.

[51] H. Hindy, D. Brosset, E. Bayne, *et al.*, "A taxonomy of network threats and the effect of current datasets on intrusion detection systems," *IEEE Access*, vol. 8, pp. 104 650–104 675, 2020. DOI: 10.1109/ACCESS.2020.3000179.

[52] J. Holland, P. Schmitt, N. Feamster, and P. Mittal, "New directions in automated traffic analysis," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 3366–3383, ISBN: 9781450384544. DOI: 10.1145/3460120.3484758. [Online]. Available: https://doi.org/10.1145/3460120.3484758.

[53] S. Jain and Y. K. Meena, "Byte level n–gram analysis for malware detection," in *Computer Networks and Intelligent Computing*, K. R. Venugopal and L. M. Patnaik, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 51–59, ISBN: 978-3-642-22786-8.

[54] P. R. M. Júnior, T. E. Boult, J. Wainer, and A. Rocha, "Open-set support vector machines," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pp. 1–14, 2021. DOI: 10.1109/TSMC.2021.3074496.

[55] D. Jurafsky and J. Martin, *Speech and Language Processing*, 2nd ed. Upper Saddle River, NJ: Pearson, May 2008.

[56] S. Khattak, N. R. Ramay, K. R. Khan, A. A. Syed, and S. A. Khayam, "A taxonomy of botnet behavior, detection, and defense," *IEEE Communications Surveys Tutorials*, vol. 16, no. 2, pp. 898–924, 2014. DOI: 10.1109/SURV.2013.091213.00134.

[57] N. Kheir, "Behavioral classification and detection of malware through HTTP User-Agent anomalies," *Journal of Information Security and Applications*, vol. 18, no. 1, pp. 2–13, 2013.

[58] N. Kheir, G. Blanc, H. Debar, J. Garcia-Alfaro, and D. Yang, "Automated classification of c&c connections through malware URL clustering," in *ICT Systems Security and Privacy Protection*, H. Federrath and D. Gollmann, Eds., vol. 455, Series Title: IFIP Advances in Information and Communication Technology, Cham: Springer International Publishing, 2015, pp. 252–266. DOI: 10.1007/978-3-319-18467-8_17. [Online]. Available: http://link.springer.com/10.1007/978-3-319-18467-8_17.

[59] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, "Survey of intrusion detection systems: Techniques, datasets and challenges," *Cybersecurity*, vol. 2, no. 1, Jul. 2019. DOI: 10.1186/s42400-019-0038-7. [Online]. Available: https://doi.org/10.1186/s42400-019-0038-7.

[60] T. Kim and H. Ju, "Effective dns server fingerprinting method," in *2011 13th Asia-Pacific Network Operations and Management Symposium*, 2011, pp. 1–4. DOI: 10.1109/APNOMS.2011.6076955.

[61] *Krajobraz bezpieczeństwa polskiego internetu. Raport roczny z działalności CERT Polska 2021*, Dostęp: 10.05.2022 r., 2022. [Online]. Available: https://www.cert.pl/uploads/docs/Raport_CP_2021.pdf.

[62] Y. Lai, G. Ping, Y. Wu, C. Lu, and X. Ye, "OpenSMax: Unknown Domain Generation Algorithm Detection," en, *Santiago de Compostela*, p. 8, 2020.

[63] P. Laperdrix, N. Bielova, B. Baudry, and G. Avoine, "Browser fingerprinting: A survey," *CoRR*, vol. abs/1905.01051, 2019. arXiv: 1905.01051. [Online]. Available: http://arxiv.org/abs/1905.01051.

[64] T. Lewis, "HTTP header heuristics for malware detection," SANS Institute InfoSec Reading Room, Tech. Rep., 2013.

[65] J. Li, C. Gu, F. Wei, *et al.*, "LightSEEN: Real-Time Unknown Traffic Discovery via Lightweight Siamese Networks," en, *Security and Communication Networks*, vol. 2021, W. Liu, Ed., pp. 1–12, Oct. 2021, ISSN: 1939-0122, 1939-0114. DOI: 10.1155/2021/8267298. [Online]. Available: https://www.hindawi.com/journals/scn/2021/8267298/.

[66] J. Li, H. Zhou, S. Wu, *et al.*, "FOAP: Fine-Grained Open-World android app fingerprinting," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022, p. 18. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/li-jianfeng.

[67] K. Li, R. Chen, L. Gu, C. Liu, and J. Yin, "A method based on statistical characteristics for detection malware requests in network traffic," in *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, Guangzhou: IEEE, Jun. 2018, pp. 527–532. DOI: 10.1109/DSC.2018.00084. [Online]. Available: https://ieeexplore.ieee.org/document/8411905/.

[68] Z. Li, H. Pan, W. Liu, F. Xu, Z. Cao, and G. Xiong, "A network attack forensic platform against HTTP evasive behavior," *The Journal of Supercomputing*, vol. 73, no. 7, pp. 3053–3064, Jul. 2017, ISSN: 0920-8542, 1573-0484. DOI: 10.1007/s11227-016-1924-3. [Online]. Available: http://link.springer.com/10.1007/s11227-016-1924-3.

[69] Z. Li, L. Sun, Q. Yan, W. Srisa-an, and Z. Chen, "Droidclassifier: Efficient adaptive mining of application-layer header for classifying android malware," in *International Conference on Security and Privacy in Communication Systems*, Springer, 2016, pp. 597–616.

[70] Z. Liu, S. Li, Y. Zhang, X. Yun, and Z. Cheng, "Efficient malware originated traffic classification by using generative adversarial networks," in *2020 IEEE Symposium on Computers and Communications (ISCC)*, Rennes, France: IEEE, Jul. 2020, pp. 1–7, ISBN: 978-1-72818-086-1. DOI: 10.1109/ISCC50000.2020.9219561. [Online]. Available: https://ieeexplore.ieee.org/document/9219561/.

[71] P. Madhyastha and R. Jain, "On model stability as a function of random seed," in *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*, Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 929–939. DOI: 10.18653/v1/K19-1087. [Online]. Available: https://aclanthology.org/K19-1087.

[72] G. Marín, P. Casas, and G. Capdehourat, "DeepMAL – deep learning models for malware traffic detection and classification," *arXiv:2003.04079 [cs]*, Mar. 10, 2020. arXiv: 2003.04079. [Online]. Available: http://arxiv.org/abs/2003.04079.

[73] S. Miller and P. Smith, "Rise of Legitimate Services for Backdoor Command and Control," Anomali, Tech. Rep., 2017, Dostęp: 27.07.2020 r. [Online]. Available: https://www.anomali.com/files/anomali-labs-reports/legit-services.pdf.

[74] S. Mizuno, M. Hatada, T. Mori, and S. Goto, "Botdetector: A robust and scalable approach toward detecting malware-infected devices," in *2017 IEEE Int. Conference on Communications (ICC)*, IEEE, 2017, pp. 1–7.

[75] M. Mokbel, *A journey into malware http communication channels spectacles*, Dostęp: 19.12.2021. [Online]. Available: https://github.com/msuiche/OPCDE/blob/master/2020/July/15/mlwr_http_comm_spect_mfmokbel_paper_v1.pdf.

[76] R. Montoro, *HTTP Header Hunter – Looking for malicious behavior into your HTTP header traffic*, Dostęp: 29.03.2019 r., 2011. [Online]. Available: http://2011.video.sector.ca/video/39786962.

[77] A. Nelson, "Sandnet++-A framework for analysing and visualising network traffic from malware," Information Security Group, Royal Holloway University of London, Tech. Rep., 2016.

[78] J. Networks, *Configuring custom messages for enhanced web filtering*, Dostęp: 19.12.2021. [Online]. Available: https://www.juniper.net/documentation/en_US/junos/topics/task/configuration/utm-ewf-custom-message-configuring.html.

[79] A. Oza, K. Ross, R. M. Low, and M. Stamp, "HTTP attack detection using n-gram analysis," en, *Computers & Security*, vol. 45, pp. 242–254, Sep. 2014, ISSN: 01674048. DOI: 10.1016/j.cose.2014.06.002. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167404814000959.

[80] R. Perdisci, D. Ariu, and G. Giacinto, "Scalable fine-grained behavioral clustering of HTTP-based malware," *Computer Networks*, vol. 57, no. 2, pp. 487–500, Feb. 2013, ISSN: 13891286. DOI: 10.1016/j.comnet.2012.06.022. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1389128612002678.

[81] R. Perdisci, W. Lee, and N. Feamster, "Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces," in *NSDI*, vol. 10, 2010, p. 14.

[82] H. Poston, *Hypertext transfer protocol (http) with wireshark*, Dostęp: 19.12.2021. [Online]. Available: https://resources.infosecinstitute.com/topic/hypertext-transfer-protocol-http-with-wireshark/.

[83] B. Qi, Z. Shi, Y. Wang, J. Wang, Q. Wang, and J. Jiang, "BotTokenizer: Exploring network tokens of HTTP-based botnet using malicious network traces," in *Information Security and Cryptology*, X. Chen, D. Lin, and M. Yung, Eds., vol. 10726, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2018, pp. 383–403. DOI: 10.1007/978-3-319-75160-3_23. [Online]. Available: http://link.springer.com/10.1007/978-3-319-75160-3_23.

[84] S. Raschka, "Model evaluation, model selection, and algorithm selection in machine learning," *CoRR*, vol. abs/1811.12808, 2018. arXiv: 1811.12808. [Online]. Available: http://arxiv.org/abs/1811.12808.

[85] P. A. A. Resende and A. C. Drummond, "HTTP and contact-based features for botnet detection," *Security and Privacy*, vol. 1, no. 5, e41, Sep. 2018, ISSN: 24756725. DOI: 10.1002/spy2.41. [Online]. Available: http://doi.wiley.com/10.1002/spy2.41.

[86] I. Rosenberg, A. Shabtai, Y. Elovici, and L. Rokach, "Adversarial machine learning attacks and defense methods in the cyber security domain," *ACM Computing Surveys*, vol. 54, no. 5, pp. 1–36, Jun. 2022. DOI: 10.1145/3453158. [Online]. Available: https://doi.org/10.1145/3453158.

[87] C. Rossow, C. J. Dietrich, H. Bos, *et al.*, "Sandnet: Network traffic analysis of malicious software," in *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, ACM, 2011, pp. 78–88.

[88] E. M. Rudd, L. P. Jain, W. J. Scheirer, and T. E. Boult, "The extreme value machine," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 3, pp. 762–768, 2018. DOI: 10.1109/TPAMI.2017.2707495.

[89] M. N. Sakib and C.-T. Huang, "Using anomaly detection based techniques to detect HTTP-based botnet c&c traffic," in *2016 IEEE International Conference on Communications (ICC)*, Kuala Lumpur, Malaysia: IEEE, May 2016, pp. 1–6, ISBN: 978-1-4799-6664-6. DOI: 10.1109/ICC.2016.7510883. [Online]. Available: http://ieeexplore.ieee.org/document/7510883/.

[90] W. J. Scheirer, L. P. Jain, and T. E. Boult, "Probability models for open set recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 11, pp. 2317–2324, 2014. DOI: 10.1109/TPAMI.2014.2321392.

[91] I. Security, *Cost of a data breach report 2021*, Dostęp: 27.03.2022 r. [Online]. Available: https://www.ibm.com/downloads/cas/OJDVQGRY.

[92] O. Segal, A. Fridman, and E. Shuster, "Passive Fingerprinting of HTTP/2 Clients," Akamai, Tech. Rep., 2017. [Online]. Available: https://www.akamai.com/uk/en/multimedia/documents/white-paper/passive-fingerprinting-of-http2-clients-white-paper.pdf.

[93] W. M. Shbair, T. Cholez, J. Francois, and I. Chrisment, *A survey of https traffic and services identification approaches*, 2020. arXiv: 2008.08339 [cs.CR].

[94] C.-S. Shieh, W.-W. Lin, T.-T. Nguyen, C.-H. Chen, M.-F. Horng, and D. Miu, "Detection of Unknown DDoS Attacks with Deep Learning and Gaussian Mixture Model," en, *Applied Sciences*, vol. 11, no. 11, p. 5213, Jun. 2021, ISSN: 2076-3417. DOI: 10.3390/app11115213. [Online]. Available: https://www.mdpi.com/2076-3417/11/11/5213.

[95] L. Shu, H. Xu, and B. Liu, *Doc: Deep open classification of text documents*, 2017. arXiv: 1709.08716 [cs.CL].

[96] M. Soltani, B. Ousat, M. J. Siavoshani, and A. H. Jahangir, "An Adaptable Deep Learning-Based Intrusion Detection System to Zero-Day Attacks," en, *arXiv:2108.09199 [cs]*, Aug. 2021, arXiv: 2108.09199. [Online]. Available: http://arxiv.org/abs/2108.09199.

[97] C. Stouffer, *10 types of malware + how to prevent malware from the start*, Dostęp: 19.12.2021. [Online]. Available: https://us.norton.com/internetsecurity-malware-types-of-malware.html.

[98] G. Stringhini, M. Egele, A. Zarras, T. Holz, C. Kruegel, and G. Vigna, "B@bel: Leveraging email delivery for spam mitigation," in *21st USENIX Security Symposium (USENIX Security 12)*, Bellevue, WA: USENIX Association, Aug. 2012, pp. 16–32. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/stringhini.

[99] S. Taheri, M. Salem, and J.-S. Yuan, "Leveraging image representation of network traffic data and transfer learning in botnet detection," *Big Data and Cognitive Computing*, vol. 2, no. 4, p. 37, Nov. 27, 2018, ISSN: 2504-2289. DOI: 10.3390/bdcc2040037. [Online]. Available: http://www.mdpi.com/2504-2289/2/4/37.

[100] Z. Tang, Q. Wang, W. Li, H. Bao, F. Liu, and W. Wang, "HSLF: HTTP header sequence based LSH fingerprints for application traffic classification," in *Computational Science – ICCS 2021*, M. Paszynski, D. Kranzlmüller, V. V. Krzhizhanovskaya, J. J. Dongarra, and P. M. A. Sloot, Eds., vol. 12742, Series Title: Lecture Notes in Computer Science, Springer International Publishing, 2021, pp. 41–54. DOI: 10.1007/978-3-030-

77961-0_5. [Online]. Available: https://link.springer.com/10.1007/978-3-030-77961-0_5.

[101] D. Ucci, L. Aniello, and R. Baldoni, "Survey of machine learning techniques for malware analysis," *Computers & Security*, vol. 81, pp. 123–147, 2019, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2018.11.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404818303808.

[102] A. Udiyono, C. Lim, and Lukas, "Botnet detection using DNS and HTTP traffic analysis," in *Proceedings of the International Conference on Engineering and Information Technology for Sustainable Industry*, Tangerang Indonesia: ACM, Sep. 28, 2020, pp. 1–6, ISBN: 978-1-4503-8771-2. DOI: 10.1145/3429789.3429818. [Online]. Available: https://dl.acm.org/doi/10.1145/3429789.3429818.

[103] N. Villeneuve and J. Bennett, "Detecting apt activity with network traffic analysis," Trend Micro Incorporated Research Paper, Tech. Rep., 2012, pp. 1–13.

[104] G. Vormayr, T. Zseby, and J. Fabini, "Botnet communication patterns," *IEEE Communications Surveys Tutorials*, vol. 19, no. 4, pp. 2768–2796, 2017. DOI: 10.1109/COMST.2017.2749442.

[105] C. Wang, B. Wang, Y. Sun, *et al.*, "Intrusion Detection for Industrial Control Systems Based on Open Set Artificial Neural Network," en, *Security and Communication Networks*, vol. 2021, C.-H. Chen, Ed., pp. 1–14, Aug. 2021, ISSN: 1939-0122, 1939-0114. DOI: 10.1155/2021/4027900. [Online]. Available: https://www.hindawi.com/journals/scn/2021/4027900/.

[106] K. Wang, G. Cretu, and S. J. Stolfo, "Anomalous payload-based worm detection and signature generation," in *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'05, Seattle, WA: Springer-Verlag, 2005, pp. 227–246, ISBN: 3540317783. DOI: 10.1007/11663812_12. [Online]. Available: https://doi.org/10.1007/11663812_12.

[107] Wei Wang, Ming Zhu, Xuewen Zeng, Xiaozhou Ye, and Yiqiang Sheng, "Malware traffic classification using convolutional neural network for representation learning," in *2017 International Conference on Information Networking (ICOIN)*, Da Nang, Vietnam: IEEE, 2017, pp. 712–717, ISBN: 978-1-5090-5124-3. DOI: 10.1109/ICOIN.2017.7899588. [Online]. Available: http://ieeexplore.ieee.org/document/7899588/.

[108] *What is a Botnet? Botnet Attack Types Defined*, Dostęp: 19.12.2021. [Online]. Available: https://www.crowdstrike.com/cybersecurity-101/botnets/.

[109] X. Wu, D. Zhang, X. Su, and W. Li, "Detect repackaged android application based on HTTP traffic similarity: Detect repackaged android application based on HTTP traffic similarity," *Security and Communication Networks*, vol. 8, no. 13, pp. 2257–2266, Sep. 10, 2015, ISSN: 19390114. DOI: 10.1002/sec.1170. [Online]. Available: http://doi.wiley.com/10.1002/sec.1170.

[110] J. Xie, S. Li, X. Yun, Y. Zhang, and P. Chang, "HSTF-model: An HTTP-based trojan detection model via the hierarchical spatio-temporal features of traffics," *Computers & Security*, vol. 96, p. 101 923, Sep. 2020, ISSN: 01674048. DOI: 10.1016/j.cose.2020.101923. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S016740482030198X.

[111] F. Xu, H. Pan, Z. Cao, *et al.*, "Identifying malware with HTTP content type inconsistency via header-payload comparison," in *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, San Diego, CA: IEEE, Dec. 2017, pp. 1–7, ISBN: 978-1-5090-6468-7. DOI: 10.1109/PCCC.2017.8280450. [Online]. Available: http://ieeexplore.ieee.org/document/8280450/.

[112] P. Xu, C. Eckert, and A. Zarras, "Falcon: Malware detection and categorization with network traffic images," in *Artificial Neural Networks and Machine Learning – ICANN 2021*, I. Farkaš, P. Masulli, S. Otte, and S. Wermter, Eds., Cham: Springer International Publishing, 2021, pp. 117–128.

[113] J. Yang, X. Chen, S. Chen, X. Jiang, and X. Tan, "Conditional Variational Auto-Encoder and Extreme Value Theory Aided Two-Stage Learning Approach for Intelligent Fine-Grained Known/Unknown Intrusion Detection," en, *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 3538–3553, 2021, ISSN: 1556-6013, 1556-6021. DOI: 10.1109/TIFS.2021.3083422. [Online]. Available: https://ieeexplore.ieee.org/document/9439944/.

[114] L. Yang, A. Finamore, F. Jun, and D. Rossi, "Deep Learning and Zero-Day Traffic Classification: Lessons learned from a commercial-grade dataset," en, *IEEE Transactions on Network and Service Management*, pp. 1–1, 2021, ISSN: 1932-4537, 2373-7379. DOI:

10.1109/TNSM.2021.3122940. [Online]. Available: https://ieeexplore.ieee.org/document/9585567/.

[115] A. Zarras, A. Papadogiannakis, R. Gawlik, and T. Holz, "Automated generation of models for fast and precise detection of HTTP-based malware," in *2014 Twelfth Annual International Conference on Privacy, Security and Trust*, Toronto, ON, Canada: IEEE, Jul. 2014, pp. 249–256. DOI: 10.1109/PST.2014.6890946. [Online]. Available: http://ieeexplore.ieee.org/document/6890946/.

[116] Y. Zhang, J. Niu, D. Guo, Y. Teng, and X. Bao, "Unknown Network Attack Detection Based on Open Set Recognition," en, *Procedia Computer Science*, vol. 174, pp. 387–392, 2020, ISSN: 18770509. DOI: 10.1016/j.procs.2020.06.104. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1877050920316240.

[117] Z. Zhang, Y. Zhang, D. Guo, and M. Song, "A scalable network intrusion detection system towards detecting, discovering, and learning unknown attacks," en, *International Journal of Machine Learning and Cybernetics*, vol. 12, no. 6, pp. 1649–1665, Jun. 2021, ISSN: 1868-8071, 1868-808X. DOI: 10.1007/s13042-020-01264-7. [Online]. Available: https://link.springer.com/10.1007/s13042-020-01264-7.

Spis rysunków

1.1 Przykład problemu otwartozbiorowego: (a) zbiór danych czterech klas; (b) granice decyzyjne standardowego klasyfikatora zamkniętozbiorowego SVC; (c) granice decyzyjne klasyfikatora otwartozbiorowego maszyny wektorów ekstremalnych EVM. Punkty danych klas nieznanych są oznaczone gwiazdką i krzyżykiem. Kształt granic decyzyjnych klasyfikatora SVC powoduje przypisanie klas nieznanych do dwóch klas znanych. Klasyfikator EVM poprawnie uznaje je za punkty klasy nieznanej. 12
1.2 Zależność wyników kolejnych kroków badań od siebie. 14
6.1 Efektywność klasyfikacji algorytmów dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger* dla wyboru hiperparametrów względem miary F1. 133
6.2 Efektywność klasyfikacji dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger* dla wyboru hiperparametrów względem miary MCC. 134
6.3 Czasy wykonania trzech faz analizy przez badane algorytmy dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger*. Wybór hiperparametrów względem miary F1. 138
6.4 Czasy wykonania trzech faz analizy przez badane algorytmy dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger*. Wybór hiperparametrów względem miary MCC. 140
6.5 Zbiorcze zestawienie wyników klasyfikacji oraz czasów wykonania analizy algorytmu EVM dla pełnej i domyślnej reprezentacji żądań HTTP narzędzia *Hfinger*. 146
6.6 Zbiorcze zestawienie wyników klasyfikacji oraz czasów wykonania analizy algorytmu W-SVM dla pełnej i domyślnej reprezentacji żądań HTTP narzędzia *Hfinger*. 148
6.7 Zbiorcze zestawienie wyników klasyfikacji oraz czasów wykonania analizy algorytmu OSSVM dla pełnej i domyślnej reprezentacji żądań HTTP narzędzia *Hfinger*. 150
6.8 Zbiorcze zestawienie wyników klasyfikacji oraz czasów wykonania analizy algorytmu MCR-SVM dla pełnej i domyślnej reprezentacji żądań HTTP narzędzia *Hfinger*. 152
6.9 Zbiorcze zestawienie wyników klasyfikacji oraz czasów wykonania analizy algorytmu RF dla pełnej i domyślnej reprezentacji żądań HTTP narzędzia *Hfinger*. 154

6.10 Porównanie wyników klasyfikacji dwóch badanych algorytmów dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger* oraz reprezentacji trigramowej (wybór hiperparametrów względem miary F1). 159
6.11 Porównanie wyników klasyfikacji dwóch badanych algorytmów dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger* oraz reprezentacji trigramowej (wybór hiperparametrów względem miary MCC). 160
6.12 Porównanie czasów wykonania faz analizy przez dwa badane algorytmy dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger* oraz reprezentacji trigramowej (wybór hiperparametrów względem miary F1). 161
6.13 Porównanie czasów wykonania faz analizy przez dwa badane algorytmy dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger* oraz reprezentacji trigramowej (wybór hiperparametrów względem miary MCC). 162
6.14 Porównanie efektywności klasyfikacji algorytmu EVM używającego reprezentacji trigramowej dla oryginalnego ziarna generatora liczb pseudolosowych oraz dwóch nowych (wybór hiperparametrów względem miary F1). 164
6.15 Porównanie efektywności klasyfikacji algorytmu EVM używającego reprezentacji trigramowej dla oryginalnego ziarna generatora liczb pseudolosowych oraz dwóch nowych (wybór hiperparametrów względem miary MCC). 164

Spis tabel

3.1 Klasyfikacja artykułów dotyczących wykorzystania ruchu sieciowego do identyfikacji pod względem scenariusza użycia. 33

3.2 Podsumowanie omówionych artykułów dotyczących wykrywania złośliwego oprogramowania przy użyciu protokołu HTTP. Użyte skróty: RF (Random Forest), SVM (Support Vector Machines), LR (Linear Regression), DT (Decisision Tree), SGD (Stochastic Gradient Descent), NB (Naive Bayes), GAN (Generative Adversarial Network), CNN (Convolutional Neural Network), LSTM (Long Short-Term Memory), GRU (Gated Recurrent Unit), OC-SVM (One Class Support Vector Machines), VPT (Vantage-Point Tree), kNN (k Nearest Neighbors), FNN (Forward Neural Network), DNN (Deep Neural Network), NN (Nearest Neighbor), LOF (Local Outlier Factor), SL (Single-Linkage). 41

3.3 Porównanie artykułów dotyczących klasyfikacji ruchu sieciowego lub złośliwego oprogramowania w scenariuszu otwartozbiorowym. 47

4.1 Źródła badań na temat analizowanych grup cech żądań. 50

6.1 Cechy żądań HTTP w dwóch reprezentacjach bazujących na aplikacji *Hfinger*. Z - to liczba zmiennoprzecinkowa, C - to liczba całkowita 119

6.2 Poziomy wartości otwartości O_P przyjęte w badaniach i odpowiadające im otwartości O^*. 127

6.3 Efektywność klasyfikacji pięciu badanych algorytmów dla pełnej reprezentacji żądań HTTP tworzonych przez narzędzie *Hfinger* dla wyboru hiperparametrów względem miary F1. 132

6.4 Efektywność klasyfikacji pięciu badanych algorytmów dla pełnej reprezentacji żądań HTTP tworzonych przez narzędzie *Hfinger* dla wyboru hiperparametrów względem miary MCC. 132

6.5 Czasy wykonania trzech etapów analizy przez badane algorytmy dla pełnej reprezentacji żądań HTTP narzędzia *Hfinger* (nazwa etapu wyboru hiperparametrów została skrócona do słowa *hiper.* dla zaoszczędzenia miejsca w tabeli). 136

6.6 Efektywność klasyfikacji pięciu badanych algorytmów dla domyślnej reprezentacji żądań HTTP tworzonych przez narzędzie *Hfinger* dla wyboru hiperparametrów względem miary F1. 144

6.7 Efektywność klasyfikacji pięciu badanych algorytmów dla domyślnej reprezentacji żądań HTTP tworzonych przez narzędzie *Hfinger* dla wyboru hiperparametrów względem miary MCC. 144
6.8 Czasy wykonania trzech etapów analizy przez badane algorytmy dla domyślnej reprezentacji żądań HTTP narzędzia *Hfinger* (nazwa etapu wyboru hiperparametrów została skrócona do słowa *hiper.* dla zaoszczędzenia miejsca w tabeli). 145
6.9 Efektywność klasyfikacji badanych algorytmów dla reprezentacji trigramowej żądań HTTP dla wyboru hiperparametrów względem miary F1 i MCC. 157
6.10 Czasy wykonania trzech etapów analizy przez badane algorytmy dla reprezentacji trigramowej żądań HTTP (nazwa etapu wyboru hiperparametrów została skrócona do słowa *hiper.* dla zaoszczędzenia miejsca w tabeli). 158

Wykaz skrótów

AE – Auto Encoder

ANN – Artificial Neural Network

ANOVA – Analysis of Variance

API – Application Programming Interface

ASCII – American Standard Code for Information Interchange

BIRCH – balanced iterative reducing and clustering using hierarchies

CAP – Compact Abating Probability

CERT – Computer Emergency Response Team

CNN – Convolutional Neural Network

CRLF – Carriage Return Line Feed

CSIRT – Computer Security Incident Reponse Team

CTU – Czech Technical University

DDoS – Distributed Denial of Service

DGA – Domain Generation Algorithm

DNN – Deep Neural Netowrk

DNS – Domain Name System

DOC – Deep Open Classification

DOI – Digital Object Identifier

DT – Decision Tree

ENISA – European Union Agency for Cybersecurity

ET – Emerging Threats

EVM – Extreme Value Machine

EVT – Extreme Value Theory

FATT – Fingerprint All The Things

FN – False Negative

FNN – Forward Neural Network

FP – False Positive

GAN – Generative Adversarial Network

GMM – Gaussian Mixture Model

GRU – Gated Recurrent Unit

HNA – Harmonic Normalized Accuracy

HTTP – Hypertext Transfer Protocol

HTTPS – Hypertext Transfer Protocol Secure

ICMP – Internet Control Message Protocol

ICS – Industrial Control System

IDS – Intrusion Detection System

IF – Impact Factor

IoT – Internet of Things

IP – Internet Protocol

IPS – Intrusion Prevention System

IRC – Internet Relay Chat

kNN – k Nearest Neighbors

LOF – Local Outlier Factor

LR – Linear Regression

LSTM – Long Short-Term Memory

MCC – Matthews correlation coefficient

MCFP – Malware Capture Facility Project

MIME – Multipurpose Internet Mail Extensions

NASK – Naukowa i Akademicka Sieć Komputerowa

NB – Naive Bayes

NIDS – Network Intrusion Detection System

NN – Nearest Neighbor

OC-SVM – One Class Support Vector Machines

OSSVM – Open-Set Support Vector Machines

PE – Portable Executable

PLOS – Positively Labeled Open Space

P2P – Peer to Peer

RAM – Random-Access Memory

RF – Random Forest

RFC – Request for Comments

SEEN – Sharpening Explanations for graph neural networks using Explanations from Neighborhoods

SGD – Stochastic Gradient Descent

SHA – Secure Hash Algorithm

SIEM – Security Information and Event Management

SL – Single Linkage

SMB – Server Message Block

SMTP – Simple Mail Transfer Protocol

SVC – Support Vector Classification

SVM – Support Vector Machines

TCP – Transmission Control Protocol

TN – True Negative

TP – True Positive

UDP – User Datagram Protocol

URI – Uniform Resource Identifier

URL – Uniform Resource Locator

US – United States

W-SVM – Weibull calibrated Support Vector Machines

USD – United States Dollar

VAE – Variational Auto Encoder

VPT – Vantage-Point Tree

Warszawa, 28.04.2022

Oświadczenie Współautorów Publikacji

Niniejszym podaję wkład autorski w publikację:

Piotr Białczak, Wojciech Mazurczyk, „Characterizing anomalies in malware-generated HTTP traffic", Security and Communication Networks, vol. 2020, pp. 1–26, Sep. 2020. DOI: 10.1155/2020/8848863

Wkład Piotr Białczaka w powstaniu publikacji obejmuje:

- sformułowanie problemu badawczego,
- przygotowanie zbiorów danych,
- stworzenie środowiska badawczego,
- przeprowadzenie badań,
- ocena wyników badań,
- przygotowanie tekstu artykułu.

Wkład procentowy Piotra Białczaka: 90%.

Imię i nazwisko współautora	Podpis współautora
Piotr Białczak	Piotr Białczak
Wojciech Mazurczyk	

Warszawa, 28.04.2022

dr hab. inż. Wojciech Mazurczyk, prof. PW
Instytut Informatyki
Wydział Elektroniki i Technik Informacyjnych
Politechnika Warszawska

OŚWIADCZENIE

Niniejsze oświadczenie dotyczy mojego wkładu w publikacji:

Piotr Białczak, Wojciech Mazurczyk, „Characterizing anomalies in malware-generated HTTP traffic", Security and Communication Networks, vol. 2020, pp. 1–26, Sep. 2020. DOI: 10.1155/2020/8848863

Mój wkład polegał na pomocy przy sformułowaniu problemu badawczego i projektowaniu badań, ocenie wyników oraz przygotowaniu tekstu artykułu.

Wojciech Mazurczyk

Warszawa, 28.04.2022

Oświadczenie Współautorów Publikacji

Niniejszym podaję wkład autorski w publikację:

Piotr Białczak, Wojciech Mazurczyk, „Hfinger: Malware HTTP request fingerprinting", Entropy, vol. 23, no. 5, 2021, ISSN: 1099-4300, DOI: 10.3390/e23050507.

Wkład Piotr Białczaka w powstaniu publikacji obejmuje:

- sformułowanie problemu badawczego,
- przygotowanie zbiorów danych,
- stworzenie środowiska badawczego,
- implementacja badanego narzędzia,
- przeprowadzenie badań,
- ocena wyników badań,
- przygotowanie tekstu artykułu.

Wkład procentowy Piotra Białczaka: 90%.

Imię i nazwisko współautora	Podpis współautora
Piotr Białczak	Piotr Białczak
Wojciech Mazurczyk	

Warszawa, 28.04.2022

dr hab. inż. Wojciech Mazurczyk, prof. PW
Instytut Informatyki
Wydział Elektroniki i Technik Informacyjnych
Politechnika Warszawska

OŚWIADCZENIE

Niniejsze oświadczenie dotyczy mojego wkładu w publikacji:

Piotr Białczak, Wojciech Mazurczyk, „Hfinger: Malware HTTP request fingerprinting”, Entropy, vol. 23, no. 5, 2021, ISSN: 1099-4300, DOI: 10.3390/e23050507.

Mój wkład polegał na pomocy przy sformułowaniu problemu badawczego i projektowaniu badań, ocenie wyników oraz przygotowaniu tekstu artykułu.

Wojciech Mazurczyk